CASTOR Usage of Prompts

ABSTRACT

A prompt *is a part of a report's definition that is bound at report execution time, instead of report construction time. When the report is executed, a user must supply the missing data. This document provides a high level discussion of how prompts are used.*

HISTORY

Date	Author	Description
28th April, 1998	Will Hurwood	Initial Version
6th May 1998	Will Hurwood	Added in example of a prompto bject
9th June,1 998	Will Hurwood	Added "Prompts in Action" chapter
15th June, 1998	Will Hurwood	Rewrote "Prompt Types" chapter to add details
17th June, 1998	Will Hurwood	Added "Using Prompts in prompts" chapter
22nd June, 1998	Will Hurwood	Locked to Incomplete,m ore info on resolution
23rd June 1998	Will Hurwood	Added 'draft' prompt to list of prompt types
15th July 1998	Will Hurwood	Modifications introduced during engineering design – numerous minor changes, most notably reversed meaning of Export and Import
21st July 1998	Will Hurwood	Closing outd etails of usage document

1. INTRODUCTION

A report is defined by defining the objects that make up the report. A required piece of functionality for Castor is that part of the definition of an object in a report can be replaced with a prompt object. When the report is executed the user is prompted to provide the missing parts of all of the prompts found in the report's objects.

Prompts have several uses:

- They enable untrained people to "specify" a report instance. A power user constructs a report using prompt objects. An untrained person is able to select and execute the report. They then answer a series of prompted questions to determine the exact parameters used to execute this report instance. The beauty of this approach is that the untrained user does not need to know how to use either the object editors, or even the object wizards, but instead answers questions posed in their own frame of reference. Thus a true QueryTone™ application is born.
- A large installation may have many users of the data warehouse who have no authority to modify the metadata in any way. Prompts give such users the ability to 'modify' a report without changing the metadata.
- A power user could use prompts as a way of executing several similar reports. The user executes the same report several times, giving different values for the prompt on each occasion.

This document is intended to serve as an introduction to Castor prompts. We intend that engineers should read this document in order to discover how prompts are designed, how they are implemented, and the reasons for the designs. We hope that this document can also be used as a usage document, but that is not its primary purpose.

1.1 PROMPT FUNCTIONALITY

The basic functionality of prompts is to "ask a question and get an answer". To make them really useful we also allow the following additional features:

- Prompts can be shared between several objects. When several objects in a report use the same prompt we default to only asking the question once.
- Each prompt has a default value. This value can be used when the report is executed offline (e.g. through the use of the scheduler). We also expect that this value will be useful to the GUI when presenting a prompt to the user.
- Depending on the type of question that a prompt asks we allow the creator of a prompt to express restrictions on the set of values that are acceptable as answers of the question. This lets us validate the answer to a prompt.
- We provide a mechanism to save the answers given to all of the prompts in a report. When a report definition is given to the report server to be executed we can specify that the saved answers are to be used. When the report server asks the GUI to resolve a prompt it will make the saved answer available for possible use by the GUI.
- The answers saved from one report can be applied to the execution of a different report.
- It is possible to specify that a saved answer to a specific prompt should be used without prompting the user to answer the prompt again. In this case no call back will be made to the GUI to answer these prompts.

1.2 OBJECTS

In this section we list the important objects used during prompt resolution.

1.2.1 Application objects

An application object is an object created by an "Agent-level" interface. Application objects are used to define reports. The principal application objects are report definitions, filters, templates and metrics.

Each of our application objects can be viewed as a collection of properties. These properties are grouped together into interfaces. An object is defined by specifying a value for each property.

Application objects contain prompt objects that then replace parts of the application object's definition. The application object records information about how the prompts that it contains should be resolved and in particular how they should be related to prompts in other application objects.

1.2.2 Prompt objects (questions)

A prompt is an entity that can be associated with an application object to properties in the application object. An application object may have multiple prompts, each replacing different properties of the object, and the same prompt may be used several times but there cannot be more than one prompt for the same property.

A prompt should be thought of as a *question*. The designer of the object supplies a question for Castor to ask to fill in the gap in the object's definition. There will be many different types of prompts. The type of the prompt corresponds to the type of the data that constitutes a valid answer to the prompt question. (E.g. a number, a string, a filter object, multiple filter-objects, a template unit etc.)

It is not enough that a prompt specifies the type of answer we expect. A prompt also contains information used to validate an answer, and a business explanation of what the question means. It may contain a default answer to use if the user declines to answer the question.

1.2.3 Resolution objects (answers)

The process of resolving prompts is the process of binding an *answer* to each prompt question in the report instance. A resolution object is a collection of answers. (Since a report will typically contain several prompts, there is not much use in having an object that can hold a single answer.)

As we execute a report instance we build up the collection of answers in the resolution object. In addition we use a resolution object to hold the list of unresolved prompts. It provides a context for resolving the report instance.

There is a one to one relationship between prompts that appear in the resolution object, and questions that need to be asked to the user. This becomes an issue when a prompt is used several times in the same report instance. The definition of the report specifies which of these instances of the prompt should be merged into a single question. The resolution object will hold one instance of the prompt for each time that the question should be asked, not for each time that the prompt appears in the report instance.

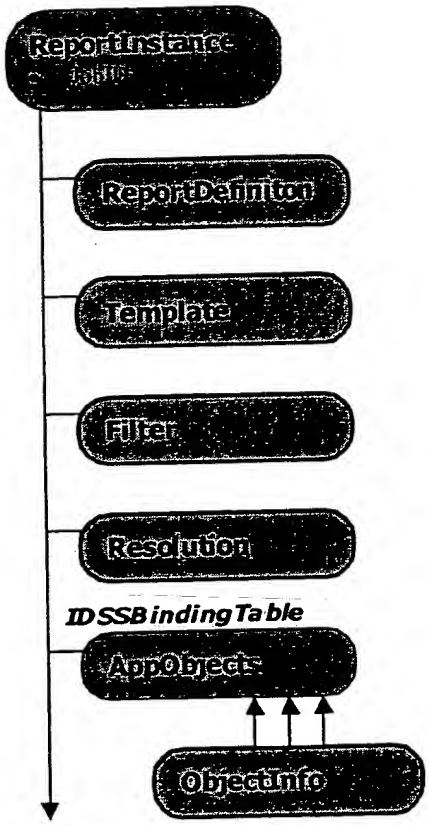
A resolution object is a persistent object. A user may save a resolution object to metadata. When a report executes we may use a saved resolution object to answer some of the prompts in the report instance. Another use of a saved resolution object is that it can be used to modify the 'default' answers to the prompts in a report.

1.2.4 Report instance

A report instance object represents a particular execution of a report. It contains the exact report definition, filter and template used for the report. It may contain an application-objects table that contains the exact version of other application objects used for this report execution.

A report instance will always contain a single resolution object. The resolution object will contain the answers given for the prompts when this report instance was executed.

The report instance object contains numerous other properties that are built up as the report is executed. Most notably this includes SQL and the result set. These properties are beyond the scope of this document, since they are not generated until after all prompts have been resolved.



One other useful property of the report instance is "Job". This returns the Job object (if known) used to execute this report instance. Internally we store the JobID. We need to store this information in a report instance, in order to match report instances with jobs during report execution.

The diagram shows the object map of the report instance. We have excluded objects not used during the resolution action of report execution.

1.3 COMPONENTS

This section lists the components used during prompt execution. Some of these "components" are really entities composed of multiple components whose details are not significant.

1.3.1 Object Server

The object server is, as always, essential. It supplies all of the DSS objects used during execution. It acts as the gatekeeper to the metadata server.

1.3.2 Metadata Server

This component stores and retrieves objects from the metadata.

1.3.3 Report Net Client

This component is used to construct and receive report execution messages. This component is used on the client side.

1.3.4 Report Net Server

This component is used to construct and receive report execution messages. This component is used on the server side.

1.3.5 The Resolution Server

This component performs the *resolution action* on a report instance. This action involves several logically distinct, but physically intertwined actions.

- It loads a version of each application object into the binding table.
- It identifies all of the prompt questions that need to be answered, and stores this information in the resolution object.
- It incorporates the user's answers to prompts by extending the binding table, and possibly identifying new prompts.

These actions are intertwined because we need to fix a version of each application object before we can identify the prompt questions, but we cannot identify which application objects are in the report instance until we have answers to the prompts. The actions have to be performed together.

1.3.6 DSS Server

This entity passes the report instance (wrapped up in a job) from component to component. The details of how it does this are beyond the scope of this document. We are only concerned with which component the report instance is delivered to next.

In a 2-tier environment the DSS Server is replaced by the ReportExecutor component. Again our scope is only where the report instance is sent to next.

1.3.7 User

The entity represents the user. From our perspective a User is a component that reads about unanswered prompts from the resolution object, and then supplies (or declines to supply) answers. The GUI interface used to communicate between the client-side report instance and a real person is beyond the scope of this document.

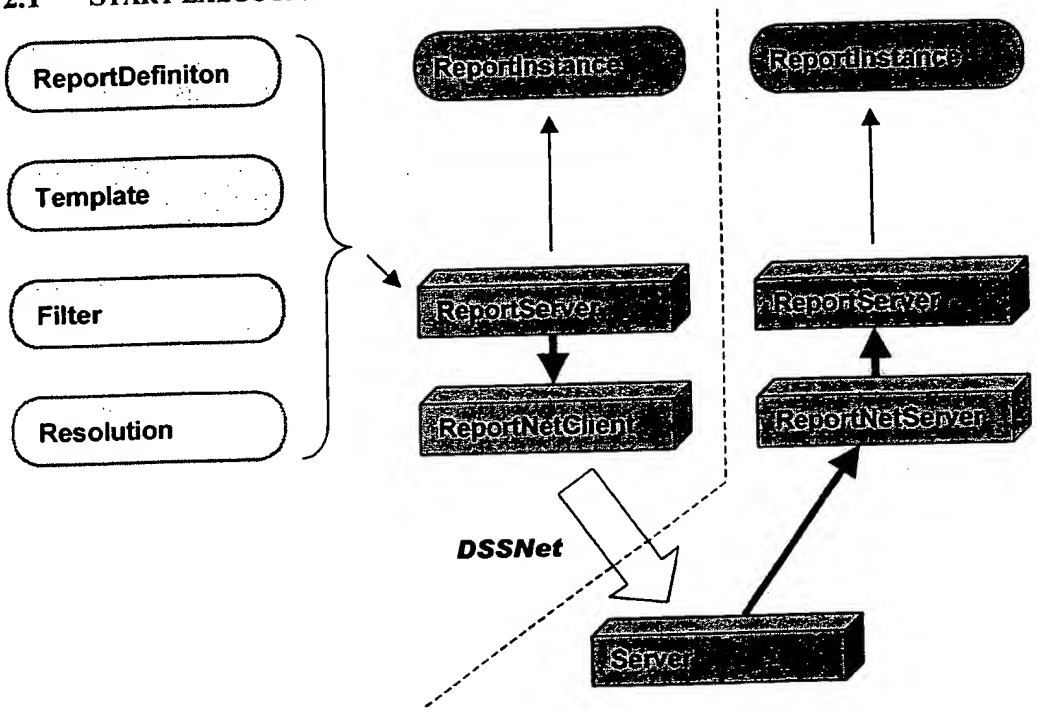
1.3.8 Other components

Nearly every Castor component can be used to execute reports. For example the element server, element net client, element net server and dB element server would all be used if the client wants to answer a 'choose elements' prompt on the client side. These components are beyond the scope of this document, since they will be invoked indirectly as they are needed.

2. FLOW OF CONTROL

The diagrams shown here show prompt execution in a three-tier environment. Two-tier execution is similar. In this case the report executor replaces the DSS Server and there are no report net objects.

2.1 START EXECUTION



This diagram shows how execution starts in a three-tier environment. We cannot execute a report until we know either the report definition, or a filter and template.

1. The client calls methods on the report server to create and execute a client-side report instance. There are several ways that this can be done. The client may supply a report definition, or a filter and template. The client may create and execute the report instance with one command, or may use two commands. The client has an option to copy an existing resolution object into the report instance. This allows the client to provide answers to some prompt questions before they are asked. We always create a client-side report instance, but if the required object definitions are not already loaded on the client, we hold off on populating it.
2. The report server calls the report net client to transfer the execution request to the server.
3. The server uses a report net server to decode the message.
4. The report server determines if we have hit the cache. If the report contains prompts we may not be able to determine whether or not there is a cache hit.
5. Assuming there was no cache hit, the report server creates a server-side report instance for this report. Control is returned to the DSS Server.

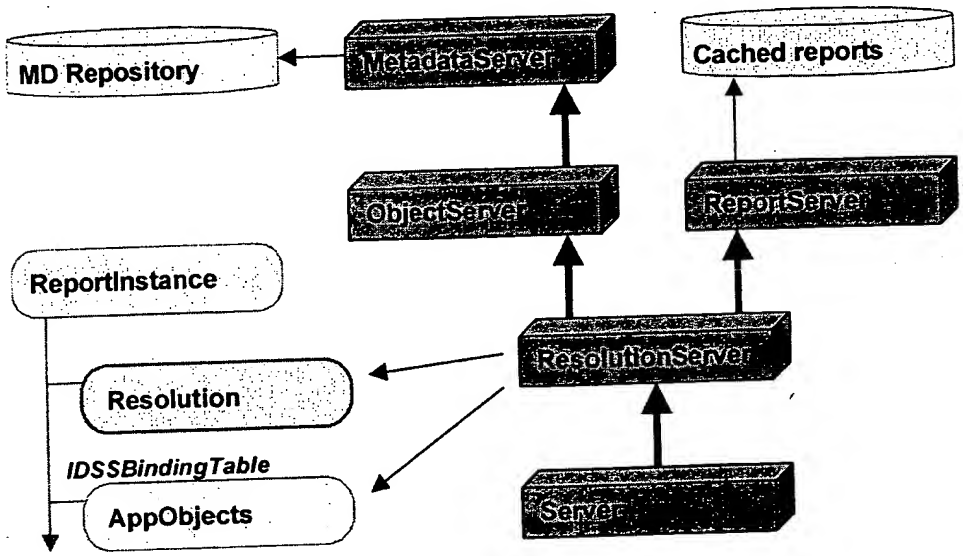
2.2 ACKNOWLEDGEMENT

The server needs to create a job to execute the report. We need to give the JobID to the client-side report instance so that the user can inquire about the job's progress.

1. The DSS Server sends a standard message back to the client.
2. The report net client interprets the message as a job acknowledgement. It writes the JobID into the client-side report instance.

From time to time the server may send progress report messages, and other messages to the client. The client handles them in a similar way to acknowledgement messages. The difference is that the client may raise an event with the user when it receives a progress report message. These messages are beyond the scope of this document.

2.3 RESOLUTION

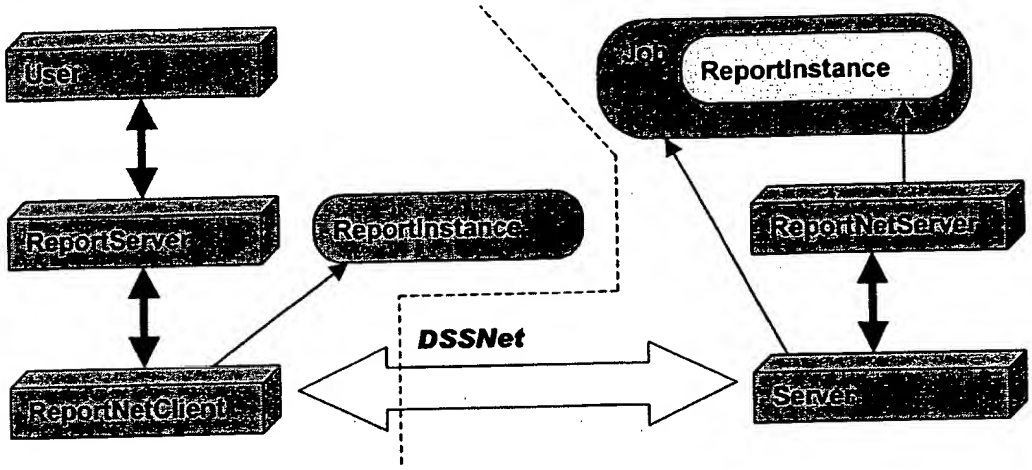


The first task is resolving the report. This is the stage in which we load the application objects into the report instance, and also we discover prompts.

1. The DSS Server passes the report instance to the resolution server.
2. The resolution server uses the object server to get current versions of the objects in the report instance. It stores references to these objects in the report instance's `AppObjects` table.
3. The object server uses a metadata server to load in missing objects (if there are any). If the resolution server was started with its 'CacheOnly' flag set, and it needs to load in additional objects, then the resolution server abandons report execution at this point and passes control back to the DSS Server.
4. The resolution server obtains the complete list of prompts in the report instance. It stores a list of prompts in the Resolution object. It detects whether or not the report instance has any outstanding prompts.

5. If there are prompts, but each one of them is answered, then the resolution server asks the report server to check its cache. Its possible that we get a cache hit now that wasn't available earlier. In either case we pass control back to the DSS Server.

2.4 PROMPTING THE USER



The DSS Server examines the report instance to determine what to do with it next. If it finds that execution is completed (or an error occurred) then it returns the report instance to the client in the usual way. Likewise the report instance may tell the server to pass it to another server.

However if there were unresolved prompts the report instance will ask the server to pass it back to the client.

1. The DSS Server passes the report instance back to the report net server. The report net server will prepare a message containing all the information the client is likely to need to answer the prompt.

2. The DSS Server will put its job into a sleep state. It will maintain the server-side report instance.

3. The report net client will add information to the client-side report instance, (and it may add objects to the client-side object server). It raises an event on the report server.

4. The report server raises an outstanding-prompts event with the user.

5. The user should resolve all outstanding prompts in the resolution object. The user returns from the event when this is done.

6. The report net client creates a return message. Only the changes in the resolution object need to be sent back.

7. The report net server amends the server-side report instance. It relies on the DSS Server to pass it the existing report instance.

2.5 ADDITIONAL PROMPTS

It is possible that the user's answers to the first prompts may cause further prompts to be triggered, or more application objects to be loaded. So the DSS Server will always follow up on a client's prompts by passing the report instance back to the resolution server.

This will continue until all prompts are resolved. (It would be advisable to keep count on the number of resolutions – just in case we get stuck in a loop.)

2.6 VIEW FROM USER'S PERSPECTIVE

In this section we describe how this flow of control appears to a user of a report. The user will not see most of these stages described earlier in this chapter.

We assume that the user has obtained a session. From the session the user has an IDSSSource interface (OS) to the DSS Object Server, and an IDSSReportSource interface (RS) to the DSS Report Server. The first interface is used to obtain objects; the second interface is used to execute reports.

2.6.1 Creation of report instance

First the user needs to create a report instance that she wants to execute. There are several ways to do this; these are described in another document. In summary the user needs to either find or create either a report definition object, or a filter and template object. The user does this with the DSS Object Server. Then the user does one of the three following actions: -

- The user calls a method (`Execute`, `ExecuteWithFilter` or `ExecuteWithTemplate`) on one of the DSS objects. This method creates a report instance, and starts executing it synchronously.

- The user creates a report instance with the IDSSReportSource method `NewInstance` or `NewInstanceFT`. The user then calls `Execute` on this instance. Parameters to the `Execute` method choose between synchronous or asynchronous execution. The advantage of this manner of executing reports is that it allows the user to execute reports containing objects that have not been saved to metadata. The user may also edit the objects in the report instance before execution starts.

- The user creates and executes a report instance with one command with the IDSSReportSource methods `ExecuteDefinition` or `ExecuteFT`. These methods are used to execute objects that already exist in the metadata. They avoid the unnecessary step of loading the definition of report objects to the client machine. Parameters to these methods choose between synchronous or asynchronous execution.

2.6.2 Prompts in report

As the resolution server finds prompts in the report instance, it accumulates them in the resolution object of the report instance. Execution will continue for as long as possible, but eventually we will need to obtain an answer for each prompt.

At this point the resolution server examines the report execution flag `DssReportDefaultAutoprompt` recorded in the report instance. If this flag is set, then we answer all of the prompts in a default manner without prompting the user. Either execution can

continue, or some prompt in the report has a default setting that requires user input to continue. In the latter case the report execution fails.

Otherwise the resolution server needs help from the user to continue. Control is passed back to the report server, which will raise the `ResolvePrompts` event to the user. This event is the only way that the report server will ask a user to answer prompts. In particular we will even raise this event in two-tier or in synchronous execution.

This code shows a simple sink for the event that asks the user to choose between taking the default action for all the prompts or aborting the execution. In general an event handler should take the resolution object from the report instance, and set values to the prompts that it contains. Most of the rest of this document describes the types of prompts and how they appear in the resolution object.

```
' Declare an event handler
Dim WithEvents gExecutionEvents As DSSReportServer

' Function to bind the eventhandler to the session's report server
Private Sub RegisterCallBack()
    If gExecutionEvents Is Nothing Then
        Set gExecutionEvents = Session.Component(DssRoleReportSource)
    End If
End Sub

' Event handler function for prompts
Private Sub gExecutionEvents_ResolvePrompts ( ByVal pInstance As IDSSReportInstance, _
        ByVal UserData As Long) As Long

    ' Ask user how to handle the prompts
    ' A better handler implementation would answer prompts from pInstance.Resolution one
    ' by one, and end up by returning 'DssPromptActionContinue'
    Response = MsgBox ( "Prompts found in report - do you want to abort(yes) or ignore?", _
        vbYesNo + vbCritical + vbDefaultButton2, "Prompts" )
    If Response = vbYes Then
        gExecutionEvents_ResolvePrompts = DssPromptActionAbort
    Else
        gExecutionEvents_ResolvePrompts = DssPromptActionDefault
    End If
End Sub
```

There are three responses to the prompt. These are to abort execution, continue execution (after answering some prompts), and the default action, which means that each prompt is answered in whatever default manner was defined when the prompt was included in the report.

2.6.3 Continuing execution after resolution

The resolution server will populate the report instance with properties that it generated from the definition and prompts. We have the following three properties: -

- `ResolvedDefiniton` – the DSS ReportDefinition object for this report instance.
- `ResolvedFilter` – the DSS Filter object for this report instance.
- `ResolvedTemplate` – the DSS Template object for this report instance.

In each case the property contains an object based on the original object (`Definition`, `Filter`, `Template`) but with substitutions made for all of the prompts. If there were no prompts then these properties are left to the original object. Objects other than these may have been substituted. As always use the version of the object from the binding table, not from the object server.

Subsequent stages of report execution use the resolved objects, not the original objects. In particular the final grid is based on the ResolvedTemplate, not on the original template.

2.6.4 Execution complete

As with constructing the report instance, the actions taken at the end of report execution are not the focus of this document. Suffice it to say execution can end in one of the following ways: -

- For synchronous execution the original execution function returns control to its caller. If an error ended execution, then it returns an error code.
- For asynchronous execution we raise an event from the report server that is executing this report execution. There are two events: `ExecutionComplete` or `ExecutionFailed`.

2.6.5 Refreshing a report instance

After execution has completed, a user can modify the report instance, and then ask to execute it again. Unless the user provides a special setting (do cross-tabulation only) the report instance will determine what changes need to be performed.

Changes to the resolved objects will cause the cross-tabulation (layout changes only) or SQL generation (contents changes) actions to be performed again. Changing the resolution object, or the original report definition, filter or template will cause the resolution stage (and hence engine, etc.) all to be performed again.

Changing the definition of one of the other objects will not be detected. The report instance does not check all of the objects that appear in it. Call `Execute` again to force the report instance to be re-executed from scratch.

3. PROMPTS IN ACTION

This section describes properties common to all prompts. We describe how prompts are placed in other application objects, and what information is stored with each instance of a prompt. We describe the algorithm used by the resolution server and how to relate the prompts in the resolution object with the prompts in the application objects. The details about different types of prompts will be covered in the next section.

The following list shows the big picture for how prompts are organized within objects, and how this relates to the resolution object. We will go on to explain each of these points in more detail in subsequent subsections.

- Each application object contains a collection of prompts. These are the prompts that appear in the application object. The same prompt may appear multiple times in an application object.
- The application object records information about how its prompts should be resolved. This information includes the order of resolution, explanation messages for the user and whether the prompt should be asked as a separate question or merged with other instances of the same prompt.
- The resolution object contains a prompt instance for each separate question that might be asked to the user. This collection can be accessed in two ways: -
    - In resolution order. This presents the prompts in the order in which they should be resolved. Each prompt instance indicates which object (or objects) it comes from.
    - Based upon an object and prompt identifier. This pattern of access is needed to determine the answer recorded for a specific prompt object.
- By default we resolve the prompts in objects without reference to the relationships between the objects. However when one application object (the *parent* or *container* object) contains a reference to another application object (the *child* or *dependent* object) we provide a way for the parent object to override the prompts in the child object. We call this the *import / export* mechanism.
    - The parent object may elect to *export* one or more of its prompts.
    - The child object may elect to import one or more of its prompts. If the child object is used in an object that exports a corresponding prompt then the resolution server will use the prompt from the parent object instead of the prompt in the child object. The child object's prompt will not be passed to the user for execution.
- An application object might specify that a prompt should automatically be closed as soon as it is detected in a prompt. This means that usually the user will not be prompted for the prompt. However it will still appear in the resolution object so that client components have a uniform way to discover the answers bound to prompts in a report instance interface. A user interface could still modify the answer to the prompt.

3.1 PROMPT OBJECT

A *prompt object* is a DSS Object (i.e. a metadata object) of type `DssTypePrompt`. The basic purpose of a prompt object is to record information about a question that a user could be asked

during the execution of a report. An object indicates that it contains a prompt by storing the ObjectID of the prompt object that represents the question that it wants to ask.

Usually we interpret multiple uses of the same prompt object to indicate that the resolution server should ask the question once, but apply the answer in several places. When this happens we say that the resolution server *merges* the two prompt instances.

However, in some cases (depending on information written into the object that contains the prompt instances) the resolution server may choose not two merge two prompts. When this occurs the resolution server will ask the user the prompt's question multiple times. If this happens we need to give the user additional information so that they can distinguish between different occurrences of the prompt. We support this by allowing the object that contains the prompt to override the prompt's `Meaning`, and `Title` properties.

3.1.1 Prompt properties

Here we list the principle properties of the prompt interface: -

- **Type**: We need to record the type of prompt. We distinguish between types of prompts by the type of data we need as the answer to the prompt.
- **Validation properties**: Each type of prompt has properties that restrict acceptable answers for this prompt. These properties are used to validate any answer that might be given. The nature of these properties depends on the nature of the question asked by the prompt.
- **Prompt properties**: Each validation property can be replaced by a prompt. This allows a user to define a prompt whose valid answers depend on a previous prompt.
- **Exports properties**: As with all types of objects that refer to objects that might contain prompts, prompt objects contain export collections to contain the prompts exported to the dependent objects.
- **Default**: Each type specific interface has a *'Default' property*. This property holds the default answer given to this prompt, if any is known. The default value can be overridden when a prompt is used in a particular object. The default value cannot directly contain prompts, but it may contain objects that contain prompts.
- **Title**: A string used to introduce the question. For example, "Revenue Limit".
- **Meaning**: An extended explanation (i.e. a string) about what the question means. For example this might be something like "The summary report will exclude all divisions whose total revenue was less than this amount."
- **Reuse**: A prompt records information on how we behave if we are presented with an instance of the prompt from a previous report execution – do we prompt the user again, and do we use the prompt's previous value or default value as the default answer?
- **Properties**: The prompt has the usual DSSProperties interface. We can use these to hold any generic properties that need to be stored for prompt objects, but which do not have any significance to a user.
- **Merge**: This Boolean property indicates if we want to merge together different instances of the same prompt into a single instance.
- **Index**: The index number of a prompt instance in its application object. This property is only needed if the same prompt object is used multiple times in the application object, in which case we need to be able to distinguish between different occurrences.
- **Importable**: True if we permit the prompt instance to be replaced by a prompt imported from the prompt's container.
- **ImportAs**: Used if we want to import a prompt instance as a different prompt object.

- **ExportsToPrompt**: Collection of exports the prompt's container into the prompt. Each export describes another prompt instance in the same application object that corresponds to a prompt inside this prompt instance.
- **Answer**: This property is only available at report execution time. It returns the value chosen by the user to answer this prompt (or the value we will use if the user declines to answer the prompt).
- **Locations**: The places where the prompt is found.
- **Closed**: True if the prompt has been answered.
- **Used**: True for a prompt that is used in the report instance. Allows a user to distinguish between prompts that are really used and prompts whose sole purpose is to store a previous answer. The latter prompts are useful if a user expects the prompt to come up later in the resolution process.
- **Incomplete**: True for a prompt that cannot be answered because its validation properties depend on another prompt.
- **Locked**: True for a prompt which has already been answered, but whose answer cannot be changed. We lock a prompt if the prompt's answer was used to build another prompt's validation properties.
- **Previous**: This property is only available at report execution time. If the user provided us with a previous resolution object, it contains the Answer for this prompt in the previous resolution object.
- **HasPrevious**: True if a prompt has a previous value.

A prompt object can be used in three different ways. The simplest way is as a DSS Object. This is the object we store in the metadata. The second way is as a prompt instance embedded in another object. This appears to be the same as the metadata object, but we allow the other object to override some of the properties. The third way is as a resolved (or to be resolved) prompt in a resolution object. In this case the resolution object fills in additional properties about the specific usage of the prompt.

The following table shows which properties are used in which place.

From Resolution Object		
From Prompt Instance		
Prompt Object		
Type Default Properties Validation properties Promptp roperties Imports properties Importable Title, Meaning Reuse	Merge Index ExportToPrompt Imports --- Overrides these properties Title, Meaning, Reuse Default	Answer Previous, HasPrevious Locations Closed Used Incomplete, Locked --- Overrides these properties Validation properties

3.1.2 Why do we use a DSS object for prompts?

Our design specifies that prompts will be implemented as first class objects, even though they will often appear to be part of a single object (i.e. the object in which the prompt is used). Here we list some of the reasons why it is a good idea to have first class prompt objects: -

1. The flow of control algorithm regularly calls for us to pass prompts from one place to another. The DSS Object is the level of granularity at which we transfer information. If a large filter contains a single, small prompt, we shouldn't have to send the filter to the client in order to answer the prompt.

2. A significant requirement for Castor prompts is that two objects can share the same prompt. Implementing prompts as separate objects enables us to share prompts with very little extra work. If prompts were part of the object that contains the prompt then it would be very hard to support sharing since we would need to find a way to have one DSS Object point to part of another object. It would also break the symmetry: which of the two objects that share the prompt actually contains the prompt.

3. Prompts as DSS Objects fit our expected business model. The average user does not know anything about our object model. All they want to see are the prompts, i.e. the questions that they need to answer – they are not interested in the objects around the prompts. So it makes sense to partition our objects in the same manner, by moving the prompt into a separate object.

4. Prompts as separate objects make it much easier to write a light client application that only is required to execute reports. The application only needs to understand how to respond to prompts – it does not need to incorporate knowledge of all the interfaces that we use to describe all of our application objects.

5. Our DSSProperties concept only applies to DSS Objects. By making prompts into DSS Objects we allow users and our GUI to define their own properties, and associate them with prompts. This lets us leverage existing code.

6. Prompts, as separate objects, are easier to implement than prompts as part of the surrounding object. We can implement the prompt in one place, instead of having to embed prompt code in all of our objects.

7. We need to assign a global ID to each prompt. When prompts are separate objects its much easier to refer to any particular prompt. We just refer to the prompt by using its ObjectID. If a prompt is part of another object we could refer to a prompt as "the fifth prompt in the filter" but if someone changes the filter, so that the number of prompts is changed, then this reference is broken. Since we want to be able to save prompt bindings we have to assign a GUID to each prompt anyway. Note that we lose this advantage if the same prompt appears in multiple places in a report, and the instances are not merged, since then we do need to provide additional information to determine which prompt we are using.

The main argument against having separate prompts is that it forces us to create a new object for each prompt. By using the concept of embedded objects (basically the COM API will embed one object inside another object) we can make the two objects act as one object, while still letting the internal code treat them as separate objects. In this way we get the best of both worlds.

3.2 USING A PROMPT OBJECT

The first stage of using prompts is to obtain the prompt object. This prompt object can be made especially for this execution, or it might reuse an existing prompt object. Our default behavior is to merge all instances of the same prompt object, so a user should reuse an existing prompt object if the user wants to ask the question that it represents only once.

```
' Make a new prompt object Year
Dim Year As IDSSPrompt
Set Year = OS.NewObject DssTypePrompt
```

If a user doesn't want to reuse the prompt object, then he or she can embed the object in whatever application object is going to use the prompt. Externally the prompt will appear to be part of the application object. However only its container, or another object in the container, can use an embedded object. So if a user embeds a prompt object inside an application object, then the user cannot import the prompt from another object, since no other object knows about the embedded object.

```
' Embed a new prompt object in a filter F
Dim P1 As IDSSPrompt
Set P1 = F.Info.Embedded.Add DssTypePrompt
```

The prompt object must be populated with information about the question that it represents. It must be provided with the type of answer that we expect. It may contain a default answer and some validation restrictions. The way that this information is presented to the user depends on the type of the prompt. We will list the types of prompts, and available validation restrictions in the next main Section.

```
' Here we show creation of a simple Long valued prompt
' A complex prompt requires far more information to set it up
With Year
    .PromptType = DssPromptLong
    .Default = 1998
    .Title = "Year"
    .Meaning = "The year at which analysis will be performed"
    .Minimum = 1980
    .Reuse = DssReusePrevious
End With
```

3.2.1 The Reuse property

The `Reuse` property is an enumeration. Its value indicates how we behave when a prompt is reused, or when we want to mark a prompt as closed. This value is stored as part of the prompt object. It can be overridden in a prompt instance that is not merged with other instances of the same prompt. In other words a prompt instance that has Merge set to 'False' can assign its own value to the Reuse property.

There are two situations in which the `Reuse` property is used. They are the following: -

- When a new prompt is detected we create an entry in the resolution object to hold the answer for the prompt. This prompt question has a Boolean property, `Closed`, that is used to indicate whether or not the user has answered the question. Normally we indicate that the user has not answered the question by setting this property to 'False'. However the setting of the `Reuse` property can be used to "auto-close" the prompt. This means that we mark the question as

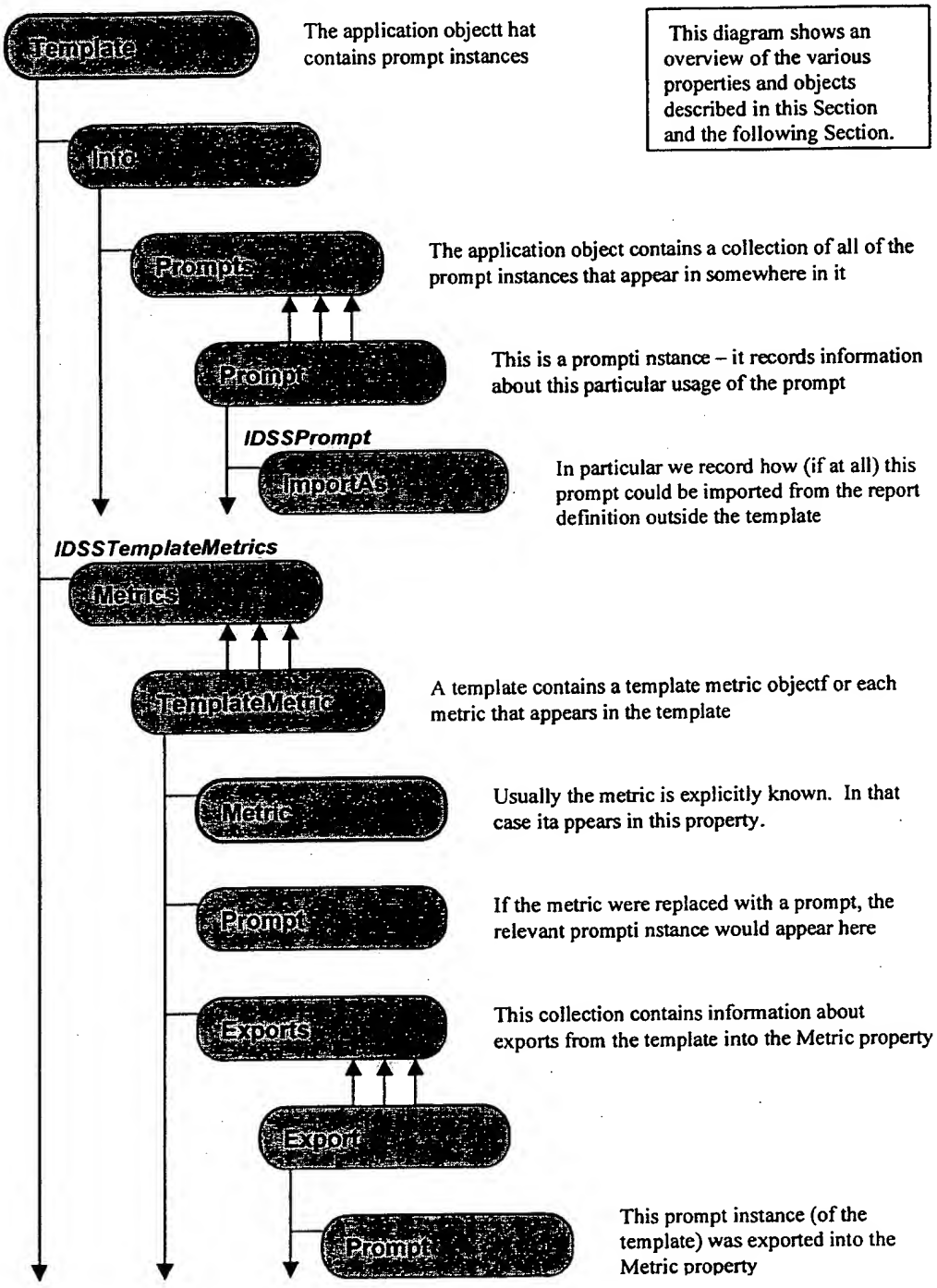
already answered as soon as we detect it. This allows us to implement a "do not ask me this question again" feature. The Reuse property tells us under what circumstances the prompt is closed automatically, and to which value it is closed.

- The Reuse property is also used to determine what to do if the user declines to answer a prompt.

The following table lists the value of the Reuse enumeration. We indicate what each value means. A prompt will not be auto-closed unless the value to which it should be auto-closed is defined. There are four possible ways to behave when we have to close a question: we can take the previous answer, the default answer, 'cancel prompt' the answer or we can break. Breaking means that the user doesn't have the option to decline to answer the prompt. We use a single enumeration for both effects so that we can disallow combinations that make no sense.

Constant		Decline effect	Auto close effect
DssReuseCancel	1	Cancel the prompt	None
DssReuseDefault (This is the default setting)	2	Use default value if available, Otherwise cancelt he prompt.	None
DssReusePrevious	3	Use previous value if available, Otherwise use default if available, Otherwise cancelt he prompt.	None
DssReuseBreak	5	Break.	None
DssReuseDefaultOrBreak	6	Use default value if available, Otherwise break.	None
DssReusePreviousOrBreak	7	Use previous value if available, Otherwise use default if available, Otherwise break.	None
DssReuseAutoCancel	9	Cancel the prompt.	Close to previous if available, Otherwise none.
DssReuseAutoDefault	10	Use default value if available, Otherwise cancelt he prompt.	Close to previous if available, Otherwise none.
DssReuseAutoBreak	13	Break.	Close to previous if available, Otherwise none.
DssReuseAutoDefaultOrBreak	14	Use default value if available, Otherwise break.	Close to previous if available, Otherwise none.
DssReuseAutoClose (Auto-close the prompt if there is a value to close it to)	17	Cancel the prompt.	Close to previous if available, Otherwise close to default if available, Otherwise none
DssReuseAutoCloseOrBreak (Auto-close the prompt, do nothing if there is no value, break if forced to close)	21	Break.	Close to previous if available, Otherwise close to default if available, Otherwise none
DssReuseForceAutoClose (Always auto-close the prompt, even if there is no value)	25	Cancel the prompt.	Close to previous if available, Otherwise close to default if available, Otherwise close to cancel prompt
DssReuseReserved	0	This value should not be used	

3.3 USING A PROMPT INSTANCE



3.3.1 Inserting a prompt into an object

Our interface definitions are enhanced by the addition of prompt-valued properties associated with each normal property (or properties) that can be replaced with a prompt. If a user wants to use a prompt instead of the normal property they assign the prompt to the prompt-valued property.

```
' Define a filter YearFilter: "Year = <Year>"
Dim YearFilter As IDSSFilter
Dim EqualsNode As IDSSOperator
Dim ConstantNode As IDSSConstant

' Construct the filter
Set YearFilter = OS.NewObject DssTypeFilter
Set EqualsNode = YearFilter.Expression.Add DssTypeOperator
Set EqualsNode.Function = FunctionEquals
Set EqualsNode.Folder.Add ( DssTypeShortcut ).Target = AttributeYear
Set ConstantNode = EqualsNode.Folder.Add ( DssTypeConstant )

' Instead of setting the constant's value, set its Prompt property instead
Set ConstantNode.Prompt = Year
```

Following this assignment the application object will create a prompt instance based on the given prompt object. The way to obtain the prompt instance is to read back the value of the prompt-valued property. The prompt instance appears to be the same as the prompt, but it is not the same object.

```
' We can read back the prompt instance from the filter
Dim PromptInstance As IDSSPrompt
Set PromptInstance = ConstantNode.Prompt
```

Generally speaking a user doesn't need to distinguish between the prompt object and the prompt instance. This is why we choose to give them the same interface. The main difference is that a user can override the `Title` and `Meaning` properties of a prompt instance without affecting the prompt object. This ability is essential if the same prompt object is used multiple times in an application object.

```
' Change the 'Meaning' string for the PromptInstance
PromptInstance.Meaning = "We will only retrieve data for this year"

' Now Year.Meaning <> ConstantNode.Prompt.Meaning
```

Most properties of the prompt instance are the same as the property on the prompt object. For example it is not possible for the prompt instance to have a different type, or a different default value to the prompt.

```
' In particular the prompt instance does not have its own Info object
' The property returns the Info object for the prompt object that the instance is based on
' So the following code moves from a prompt instance to the prompt object
Dim PromptObject As IDSSPrompt
Set PromptObject = PromptInstance.Info.Special
```

3.3.2 The prompt instance collection

Each application object maintains a collection of all of the prompts that are being used in the application object. Whenever a user adds a new prompt object to the application object we automatically extend the collection to contain the new prompt object. The collection returns the prompt instance, not the prompt object.

```
' Here is another way to get the prompt instance from the filter
Set PromptInstance = YearFilter.Info.Prompts.Item ( 1 )
```

The collection is ordered. The prompt instance populates the `Index` property to show a user where each instance is in the collection. The user can reorder the collection by writing to this property. The order of the instances in the collection is significant, since we will present the prompts to the user in the same order that they appear in this collection.

```
' Make another prompt object Rate: Double prompt
Dim Rate As IDSSPrompt
Set Rate = OS.NewObject DssTypePrompt
With Rate
    .PromptType = DssPromptDouble
    .Default = 1.05
    .Title = "Interest Rate"
    .Meaning = "Simple intested is added at this rate every year since 1980"
    .Minimum = 1.0
    .Maximum = 2.0
End With

' Define a metric SimpleInterest: "(<Year> - 1980) * <Rate> * Loan"
Dim SimpleInterest As IDSSMetric
Dim MultNode As IDSSOperator
Dim MinusNode As IDSSOperator

' Construct the metric
Set SimpleInterest = OS.NewObject DssTypeMetric
Set MultNode = SimpleInterest.Expression.Add DssTypeOperator
Set MultNode.Function = FunctionMultiply
Set MinusNode = MultNode.Folder.Add DssTypeOperator
Set MinusNode.Function = FunctionMinus
Set MinusNode.Folder.Add ( DssTypeConstant ).Prompt = Year
Set MinusNode.Folder.Add ( DssTypeConstant ).Value = 1980
Set MultNode.Add ( DssTypeConstant ).Prompt = Rate
Set MultNode.Add ( DssTypeShortcut ).Target = AggMetricLoan

' Set the Index property of Rate prompt instance to 1
' This will move it to the front of the collection, so Rate will be prompted before Year
MultNote(2).Prompt.Index = 1

' The following line will return an ERROR - Index only works on prompt instances
Rate.Index = 1 ' ERROR - how do we know which instance of Rate to move to the front?
```

If the user inserts a prompt object into the application object that it is already using our default behavior is to give the user the existing prompt instance, not to make a new one. This means that by default if a user puts the same prompt object in an application object in several places, we will only ask the question on one occasion.

```
' Change the metric so it contains another Rate
Set MultNode.Add ( DssTypeConstant ).Prompt = Rate

' The following line will show that the metrc still only has two prompt instances
' If execute we will prompt for Rate once, but use it twice in the metric
MsgBox "Number of prompts on metric = " & SimpleInterest.Info.Prompts.Count

' Get rid of the second usage of Rate
Call MultNode.Remove ( 4 )
```

If a user wants to have several instances of the same prompt object in an application object, then the user must first directly insert a second instance of the prompt into the collection. Then the user must assign the prompt-valued property with whichever instance (drawn from the collection) the user wants to use.

```
' Force the metric to contain a new instance of the Rate prompt
Dim RateInstance2 As IDSSPrompt
Set RateInstance2 = SimpleInterest.Info.Prompts.Add ( Rate )

' Now add this new instance of Rate to the metric
Set MultNode.Add ( DssTypeConstant ).Prompt = RateInstance2

' If the metric is executed now we will prompt twice for Rate

' Get rid of the second usage of Rate (it will automatically go from Prompts collection)
Call MultNode.Remove ( 4 )
```

3.3.3 The Merge property

The prompt-instance collection cleanly handles multiple use of prompt objects within a single application object. It indicates which order the prompts should be presented to the user, and whether or not the same prompt should be asked twice.

There remains the question of whether or not to merge instances of the same prompt object that occur in different application objects. The prompt-instance collection is no help here, since it only contains prompt instances from a single application object. As usual our default behavior is to merge instances of the same prompt object.

However we support a Boolean property called `Merge`. This property is stored at the prompt instance level. The resolution server will merge together all prompt instances (of the same prompt object) that it finds that have `Merge` set to 'True'. They will be presented to the user as a single prompt to resolve. If this property is set to 'False', for a particular prompt instance, then the resolution server will not merge that particular prompt instance in with other prompt instances (of the same prompt object) found on other application objects.

If `Merge` is set to False, we will still only prompt once for each distinct application object that contains a prompt instance. We do this even if the application object appears several times in a report. If a designer wants to use an application object twice in a report, and prompt the user twice, once for each usage, then the designer must use the prompt import / export mechanism described below.

It doesn't make sense for a client to set `Merge` to false unless the user also overrides the `Title` and `Meaning` properties. Otherwise a keyboard operator will have no way to distinguish between different instances of the same prompt object. This is why we do not permit a user to set `Merge` to False on the prompt object itself – we want the designer to explicitly choose to set `Merge` to False

on a prompt instance by prompt instance basis. If Merge is 'True' on a prompt instance we will not permit a designer to modify the Meaning and Title properties.

```
' We want the Year on YearFilter to be prompted separately from other Year instances
' We have already given this prompt instance its own Meaning
' Note: this Item call returns the first prompt based on Year in the collection
YearFilter.Info.Prompts ( Year ).Merge = False
```

Note that there appears to be a contradiction between our assertion that we merge all instances of a prompt object with Merge equal to True, and that if a prompt object appears twice in a prompt-instance collection then we will prompt the user twice for these prompts. The contradiction is resolved because we do not permit two distinct instances of the same prompt object in the same application object to both have Merge equal to True. We maintain this restriction by using the following rules: -

- The first instance of a prompt object defaults to Merge = True.
- Subsequent instances (of the same prompt object) default to Merge = True if all of the existing instances have Merge = False. But if an existing Instance has Merge = True, then the instance has to set Merge = False.
- If a client sets Merge to True, and we find an existing prompt instance of the same prompt object, in the same collection with Merge set to True, then we set the other instance's Merge to False. (Like in a radio buttons control.)
- If a client sets Merge to False we take no additional action. There is no contradiction in not having any instance with Merge set to 'True'.

3.3.4 Other prompt instance properties

In this Section we list all of the properties of the Prompt object that can be modified at the level of the prompt instance. Observe that all of the prompt object level properties can be accessed from a prompt instance, but in general they cannot be modified from a prompt instance. We do not let a prompt instance modify the object info properties, prompt type, validation restrictions or default value of a prompt object – we consider these to be fundamental properties of the prompt object. If a designer wishes to vary them, she should create a new prompt object.

We have already stated that the designer may override Title and Meaning properties on a prompt instance, so that a human user can distinguish between different instances. The Index property doesn't make any sense until we have a prompt instance collection. The Merge property is available at the prompt-instance level to describe a simple way to merge instances together.

We permit a designer to override the Reuse property at the prompt instance level. It is plausible that a designer wants to control how a particular prompt instance is reused. However it makes no sense to modify this property unless Merge is set to 'False'. If Merge is' True' we not permit a user to set this property. This is because there is no clean way to decide which of several values taken from merged prompt instances should be the one that we use.

We permit a designer to override the Default property on a prompt instance. Again it makes no sense to do this unless Merge is 'False'. This means that when a prompt question is made from this instance we will use the default value from the instance, instead of the value from the prompt object.

A designer cannot embed a prompt instance in the default answer to a prompt; nor can she export prompt instances into objects in the default answer. However the default can contain prompts.

A designer can suppress a prompt instance by forcing it to have a particular value. The designer does this by setting the `Default` property of the prompt instance to the desired value, and the `Reuse` property of the prompt instance to `DssReuseForceAutoClose`. When the report is resolved the resolution server will automatically close the prompt instance, using the local default value if it cannot find a previous value.

At first sight there is no reason for a designer to insert a prompt in an object, and also force the prompt to be closed – after all the designer could have just as easily not used a prompt at all. But this functionality is useful if the designer wants to reuse an existing object, but suppress a prompt from the object. The designer merely ensures that the prompt she wishes to suppress is merged with another prompt instance whose `Reuse` property is set in this way. This functionality is why we permit a designer to insert a prompt instance in an application object's collection, even if the prompt is not used elsewhere in the application object.

Last of all we have several properties: `Importable`, `ImportAs` and `ExportsToPrompt` that are used to implement the import / export functionality. These properties are described in the next section.

3.4 IMPORTS AND EXPORTS

For most situations the prompt instance collection suffices. However there are situations in which a user wants to have finer control over how prompt objects are resolved. For example he might want to specify that a prompt on a child object is resolved before a prompt on a parent object, or that there is a more complex merging structure than the one shared instance, and multiple singleton instances structure provided by the `Merge` property.

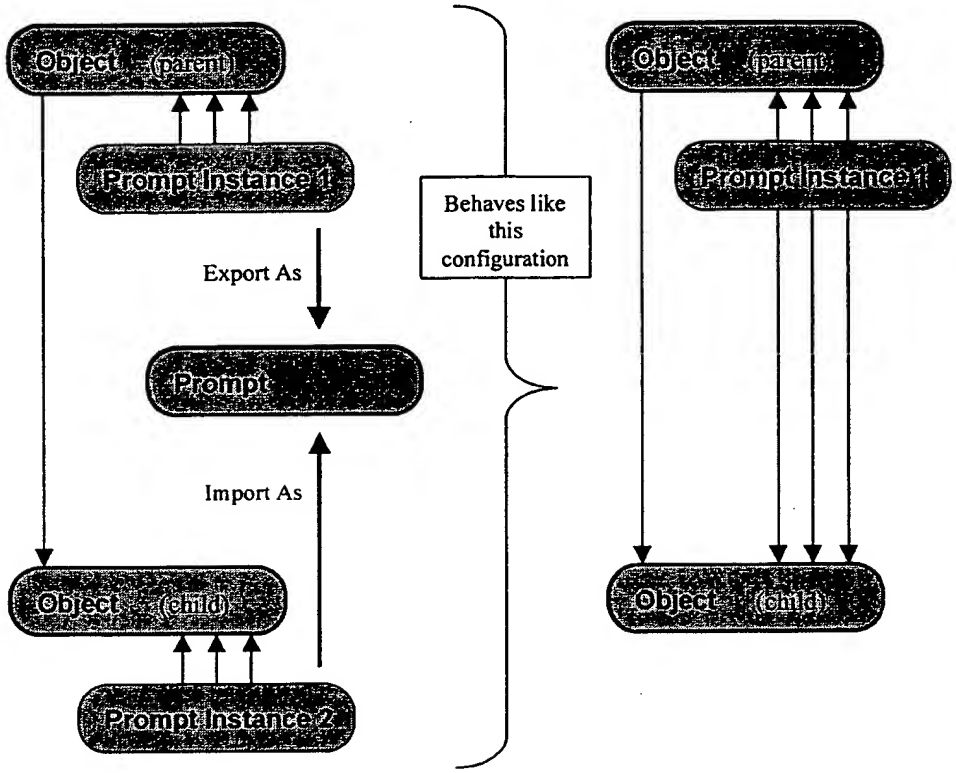
We address this by offering an import / export capacity. An object can *import* prompts from another object that uses this object as one of its dependents. An object can *export* prompts into its dependent objects. The effect of this operation is to merge prompt instances between the parent and the child. The user is prompted with the prompt instance from the parent. The answer to this instance is used as the answer of both the parent and the child prompt instance.

The import / export mechanism provides us with a way to explicitly merge prompt instances between a parent and a child object. By being selective about which prompt instances in the parent are merged with which instances in the child, a user can impose a complex merging structure. Since we resolve prompts from parents before children a user can impose a different ordering on prompts from children by importing them all to the parent. The order in which the prompts are resolved in the parent then takes priority.

We emphasize that when a prompt instance is exported from a parent object to a child object, the prompt in the child is replaced with the prompt from the parent. In particular the properties of the child prompt instance are ignored on the grounds that the user is never prompted with the child prompt instance. For example the user will be prompted with the `Title` and `Meaning` of the parent prompt instance, not of the child prompt instance.

The following diagram shows this situation. The designer specifies a report with two objects; each object with its own prompt instance. The designer also specifies that the parent object exports its instance and the child object imports its instance. The resolution server will determine that the imports and exports match. It then discards the prompt instance of the child object, and instead

applies the prompt instance of the parent object to both the parent and the child. Any settings made specifically on the child prompt instance will be lost.



3.4.1 Importing a prompt

Each application object contains a collection of prompt instances. Each prompt instance on the application object has a Boolean property called `Importable`. This property defaults to 'False'. If the designer sets this property to 'True' for some prompt instance then the resolution server will consider importing another prompt instance into the application object to replace the prompt instance.

The `Importable` property indicates that the prompt instance might be replaced by a prompt instance imported from a parent application object. However it does not require that this happens. We will not return an error if the application object's parent neglects to export a suitable prompt instance. This would merely mean that the parent was not interested in overriding the prompt's default behavior. The `Importable` property says the prompt could be imported, it does not say that it must be imported.

```
' Permit all of our example prompts to be imported
YearFilter.Info.Prompts ( 1 ).Importable = True
SimpleInterest.Info.Prompts ( 1 ).Importable = True
SimpleInterest.Info.Prompts ( 2 ).Importable = True
```

3.4.2 Exporting a prompt

An application object can export prompt instances from its collection of prompt instances into one of its dependent application objects. We do this by using a collection called `Exports`. Each dependent object (which could contain prompts) will have its own collection. If the same child object appears as a dependent in two different places inside the parent object then there will be two collections, one for instance of the child object. A designer may export different prompts into each instance of the child object.

The `Exports` collection for a child object will always appear in the same interface as the child object[1]. The collection records information about which prompt instances from this object are exported into the child object. We use a collection because we could export several prompts into the same child.

The name of the collection property will be `Exports` if an interface only contains one child object into which we could export a prompt. For example the IDSSTemplateMetric interface can only export prompts into its metric property. However an interface that contains several properties XXXX and YYYY which might contain prompts will also have properties `XXXXExports` and `YYYYExports` to contain exports into each object. For example the prompt object itself contains several properties (`Minimum`, `Restriction` etc.) whose value might contain prompts. So it also has corresponding properties `MinimumExports`, `RestrictionExports` etc.

A member of an `Exports` collection is an object with interface IDSSExport. The main property of this interface is a prompt instance property called `Prompt`. The designer sets this property to record which prompt instance is being exported.

```
' Create a template containing the metric SimpleInterest
Dim T As IDSSTemplate
Set T = OS.NewObject DssTypeTemplate
T.Rows.Add ( DssTemplateMetrics ).Special.Add ( SimpleInterest )

' Export the prompt Year into SimpleInterest
' This command will automatically add a prompt instance for Year to the template
Call T.Metrics ( SimpleInterest ).Exports.Add ( Year )
```

As with the `Importable` property, exporting a prompt instance into a dependent object merely indicates a willingness to export the prompt, not a requirement to export the prompt. The prompt will only be exported if the dependent object also imports it. The resolution server will not raise an error if it does not find a suitable import.

3.4.3 Matching imports and exports

With regard to the import / export functionality, the resolution server's task is to match the imports into a child object with the exports from its parent. When it finds a match it uses the prompt in the parent object instead of the prompt in the child object. This section describes how the resolution server determines whether or not two prompt instances match.

We match imports and exports together by comparing the ObjectID of the imported and exported prompt instances. In other words they match if they are both instances of the same prompt object. The advantage of this approach is that it allows a designer to add or remove prompt instances (based

[1] From a programmatic viewpoint it would make more sense to introduce a collection on the other object. Butt his would greatly complicate implementation for very little gain,s ince itw ould force us to create a wrapper every time one object appears in another object.

on other prompt objects) from either the parent or the child object without breaking the import / export relationship.

However there are two problems with this approach. Both of these problems are dealt with by adding extra properties to the prompt object, or to the Exports collection. Unless a designer is faced with one of these problems she can ignore these properties: -

- The designer might want to match together two instances that are based on different prompt objects. We alleviate this problem by permitting both the importer and the exporter to 'rename' the prompt object. This means that the designer can specify any prompt object that the resolution server should use in place of the prompt instance when matching imports and exports.

- The designer might want to import two different prompt instances based on the same prompt object. We alleviate this problem by allowing the designer to assign an ExportIndex number to each export.

For the first case we use prompt-object properties called ImportAs and ExportAs to 'rename' a prompt instance. These properties default to the prompt object that underlies the prompt instance.

When the designer indicates that a prompt instance could be imported by setting Importable to 'True' she may also set the ImportAs property of the prompt instance. This property's value is a prompt object (not a prompt instance). By specifying a different prompt object, the designer can make the application object represent the prompt instance as a different prompt for import purposes. The designer can set this property to any prompt object that has the same prompt type as the prompt instance.

```
' Suppose that PromptDouble is a prompt whose type is compatible to DssPromptDouble
' Then we can say that SimpleInterest imports its Rate prompt using PromptDouble
Set SimpleInterest.Info.Prompts ( Rate ).ImportAs = PromptDouble
```

For exporters the designer 'renames' the prompt by setting the ExportAs property on the IDSSExport interface. As with ImportAs this property is used to make the resolution server represent the exported prompt using a different prompt object. As with imports this parameter can only be used to switch between prompts with the same type.

```
' Export the Rate prompt from SimpleInterest to the Template
' Since it was imported as PromptDouble, we must export it the same way
Set T.Metrics ( SimpleInterest ).Exports.Add ( Rate, PromptDouble )

' The extra parameter to Add set "Exports( 1 ).ExportAs" to PromptDouble
```

We address the problem of multiple exports keyed to the same prompt object by using an index number. In fact the resolution server identifies each import and export by assigning a key consisting of a prompt object and a number to the import or export. The ImportAs and ExportAs properties are used to change which prompt object it uses. Usually the index number is 1.

The designer can change the index number associated with an export by setting the ExportIndex property of the IDSSExport interface. This property also defaults to 1. The designer can set it to any positive index number.

```
' Make the Rate export as the second PromptDouble imported from SimpleInterest
T.Metrics ( SimpleInterest ).Exports ( 1 ).ExportIndex = 2
```

The designer cannot modify the index number associated with a prompt import. The first prompt instance in the prompt instance collection that is `Importable` and whose `ImportAs` property is a particular prompt object is given the index number 1. The second importable prompt instance, imported as the same prompt object is given the number 2, and so on. Thus every importable prompt instance is assigned a different key.

Within an `Exports` collection, the same prompt instance may be exported several times with a different key each time. This merely means that several prompts in the child object will all be replaced with the same prompt from the parent object. However we will not permit several prompts to be exported with the same key, because each prompt in the child must have a unique prompt that is imported to replace it.

3.4.4 Prompts inside prompts

Prompts are application level DSS Objects. So a prompt may also contains a prompt collection, and prompt instances of its own. These work in the same way as usual. There are two reasons to use a prompt instance inside a prompt: -

- We want to use the answer of one prompt as part of the validation parameters of another prompt.
- We want to define the prompt as a structure of ordinary objects with prompts embedded inside the structure. We call this a *draft prompt*. For example a prompt can be defined as an expression containing other prompts.

The prompts that a prompt contains are part of the prompt object. They cannot be modified when a prompt object is used in a prompt instance. A prompt object can also import and export prompt instances in the usual manner. For example we can mark a prompt instance inside a prompt as `Importable`.

```
' Insert, not very usefuly, a Rate prompt in Year. Mark it as importable
Year.Info.Prompts.Add ( Rate ).Importable = True
```

Within the definition of a prompt object we can export prompt instances from the prompt object's instance collection to other application object objects that are dependents of the prompt. We do this in exactly the same way we would export an application object from any other application object to one of its dependents; we use an `Exports` collection. There are four collections accessible directly from the prompt object: `MaximumExports`, `MinimumExports`, `OriginExports` and `RestrictionExports`. There might be other export collections exposed from expressions or other sub-objects exposed from within the prompt.

However we have a special mechanism to allow any application object (including a prompt object) to export a prompt instance from itself into one of its own prompt instances. We don't want to have an `Exports` collection for each time the prompt instance is used, since: -

- The prompt instance might be used in multiple places.
- A prompt instance might not be used at all except to be exported into a dependent object.
- It would unnecessarily force us to add several more `Exports` properties.

Instead we have a collection exposed from the prompt instance itself. This collection is called `ExportsToPrompt`. It is an IDSSExports collection other prompt collections; its only difference is that it is exposed from the object into which the exports are made, instead of being exposed next to the object into which the exports are made.

The following piece of code shows how this functionality is used. We have just amended the Year prompt so that it contains a prompt instance based on the Rate prompt. We now want SimpleInterest to export its Rate prompt instance into its Year prompt instance.

```
' Add an export so that SimpleInterest:Rate is the same as SimpleInterest:Year's Rate
SimpleInterest.Info.Prompts ( Year ).ExportsToPrompt.Add ( Rate )
```

A designer cannot export a prompt into a prompt instance if the prompt instance that we are exporting into has its `Merge` property set to 'True'. This is because the designer might make several conflicting exports into different instances of the same prompt object, that all have `Merge` set to 'True'. When the resolution server merges these prompt instances, it doesn't know which exports to use. Rather than have it choose arbitrarily, or ignore the exports, we forbid exports into global prompts.

3.5 MULTIPLE ANSWERS FOR THE SAME PROMPT

Our default behavior if a prompt object occurs twice is to merge the different instances of the same prompt. A user can suppress this behavior by using the `Merge` property. If `Merge` is set to 'False' then each prompt instance (i.e. specific usage of a prompt in a specific object) will be prompted separately. However even if some application object appears twice in the report, the prompts in the object will only be asked once, and the answer will be used every time the application object is found in the report. It is not possible to use the `Merge` property to specify that a prompt instance should be asked twice, once for each occurrence of the object that contains the prompt instance.

However this effect can be achieved by using the import / export facility. We do it be specifying that the application object imports its prompt instance. Each time the application object is used in the report we can export a prompt into the application object. Provided that we export different prompt instances into each usage of the application object (and assuming that `Merge` is 'False' on these instances) they are now perceived as separate questions, and so they are not merged together.

The following example shows how this can be done. Suppose that the Year prompt is defined as before, to prompt for a single integer value. Let us define the metric "SimpleInterest2" to prompt twice for a Year. The first time it prompts for the stop year, the second time for the start year. For simplicity of this example we will fix the rate to 5% simple interest per year. The following code shows how to define this metric.

```
' Define a metric SimpleInterest2: "(<Year:1> - <Year:2>) * 1.05 * Loan"
Dim SimpleInterest2 As IDSSMetric

' Construct the metric
Set SimpleInterest2 = OS.NewObject DssTypeMetric

' We want to insert the Year prompt twice into the metric
Dim YearStop As IDSSPrompt
Dim YearStart As IDSSPrompt
Set YearStop = SimpleInterest2.Info.Prompts.Add Year
Set YearStart = SimpleInterest2.Info.Prompts.Add Year
```

```
' Define special properties for our two prompt intstances
With YearStop
    .Merge = False
    .Importable = True
    .Meaning = "The year at which we stop calculating interest"
End With
With YearStart
    .Merge = False
    .Meaning = "The year from which we start calculating interest"
End With

' Set the definition of the metric
Set MultNode = SimpleInterest2.Expression.Add DssTypeOperator
Set MultNode.Function = FunctionMultiply
Set MinusNode = MultNode.Folder.Add DssTypeOperator
Set MinusNode.Function = FunctionMinus
Set MinusNode.Folder.Add ( DssTypeConstant ).Prompt = YearStop
Set MinusNode.Folder.Add ( DssTypeConstant ).Value = YearStart
Set MultNode.Add ( DssTypeConstant ).Value = 1.05
Set MultNode.Add ( DssTypeShortcut ).Target = AggMetricLoan
```

Now we define a template "T2" and put SimpleInterest2 in its metrics collection twice. If we expected a report at this point the report's result would show the metric twice. Because we turned off the merge functionality, execution would prompt for Year twice – once for YearStart and once for YearStop. However we would only prompt once for YearStart and once for YearStop, even though the metric that contains these instances appears twice in the template. This is because our default behavior is always to prompt as rarely as possible.

```
' Create a template T2 containing the metric SimpleInterest2 twice
Dim T2 As IDSSTemplate
Set T2 = OS.NewObject DssTypeTemplate
Call T2.Rows.Add ( DssTemplateMetrics ).Special.Add ( SimpleInterest2 )
Call T2.Metrics.Add ( SimpleInterest2 )
```

Now suppose that we want to prompt separately for the stop year for each metric, but only prompt once for the start year for both metrics. We can do this by exporting the stop year to both metrics, from different prompt instances in T2. This export distinguishes between the two instances, because they are different instances in T2. So we now prompt for them separately.

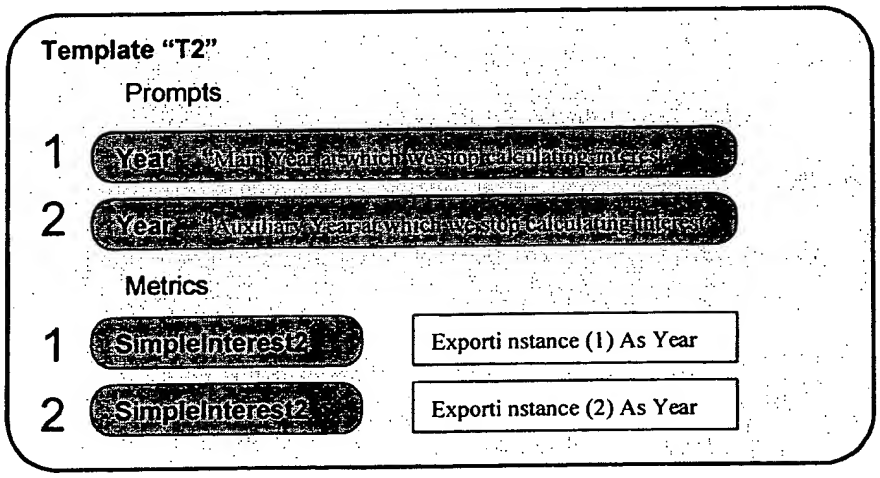
The settings of the prompt instances in T2 are used, rather than the settings of the single YearStop prompt instance on SimpleInterest2. This allows the designer to give them separate `Meaning` and `Title` strings. These separate strings are essential, because without them the user would not be able to determine which instance of the prompt she was being asked to answer.

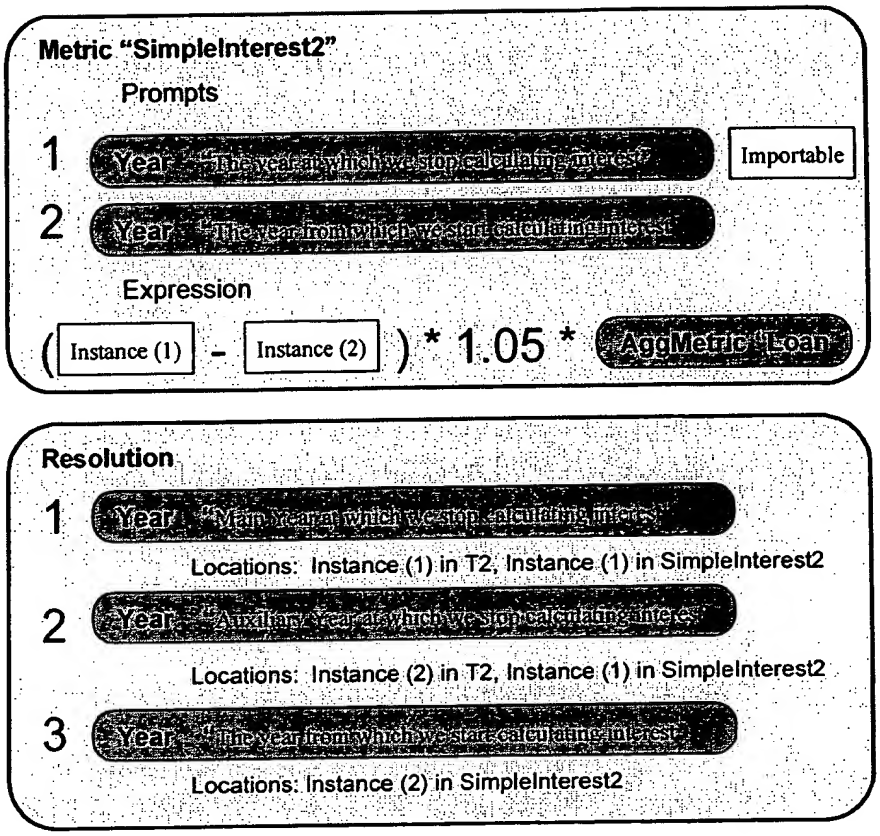
```
' We want to insert the Year prompt twice into the template
Dim YearStop1 As IDSSPrompt
Dim YearStop2 As IDSSPrompt
Set YearStop1 = T2.Info.Prompts.Add Year
Set YearStop2 = T2.Info.Prompts.Add Year

' Define special properties for our two prompt intstances
With YearStop1
    .Merge = False
    .Meaning = "Main Year at which we stop calculating interest"
End With
With YearStop2
    .Merge = False
    .Meaning = "Auxilary Year at which we stop calculating interest"
End With

' Export each local prompt instances into a different instance of the metric
T2.Metrics(1).Exports.Add YearStop1
T2.Metrics(2).Exports.Add YearStop2
```

The following diagram shows the picture when a report containing this template is resolved. The prompt Year will be resolved to three prompt questions in the resolution object, distinguished from each other as shown in the diagram.





This template still has the flaw that it will offer the prompts in the order YearStop1, YearStop2 and then YearStart. If we want to prompt for the YearStart first (which would probably make more sense to an analyst), we must also import it to T2 so that all three prompts are listed as prompt instances of the same object. We can now control the order of the prompts by ordering the prompt instances that appear in T2.

```
' We need to change SimpleInterest2 so it imports both instances of Year
YearStart.Importable = True

' We need to create a new instance of Year on T2
' (The default action would be to reuse the first exisiting instance)
Dim YearStartT2 As IDSSPrompt
Set YearStartT2 = T2.Info.Prompts.Add Year

' Need to export YearStartT2 to each occurance of the metric
' We have to do this twice, to indicate that each time the metric appears on the template
' we want to use the YearStartT2 prompt instance to answer the metric's prompt
' The 2 parameter is needed to indicate that we want to export YearStartT2 to the second
' 'Year' instance that the metric imports
Call T2.Metrics(1).Exports.Add ( YearStartT2, , 2 )
Call T2.Metrics(2).Exports.Add ( YearStartT2, , 2 )

' Finally we need to specify that YearStartT2 should be the first prompt instance in T2
YearStartT2.Index = 1
```

Observe that the import / export mechanism allows the designer to completely specify how prompts are merged and ordered. But the more control the user wishes to exercise, the more work the designer has to perform to specify what they want to do. We always default to asking as few prompts as possible.

3.6 THE RESOLUTION ORDER

In this section we describe how the resolution server determines which order to present the prompt questions to the user. We start off with a list of prompt instances, taken from the prompt instance collections of all of the objects that appear in the report instance.

The `Exports`, `Importable` and `Merge` parameters specify exactly which of these instances should be merged together to make a single prompt question. The question remains how do we know which order to present these merged objects to the user.

The answer is that we construct a list of all of the prompt instances by following this procedure.

1. Order all of the non-prompt application objects in the report instance. These go in a topological order. This means that every object appears before its dependents. In particular the report definition is always the first object. Each object only appears once in the list. We do not specify which of two unrelated objects will come first.

2. Expand this list by replacing each object, with the list of prompt instances that it contains.

3. Insert immediately before each prompt instance, the list of prompt instances that are contained in the prompt instance. It is impossible to resolve a prompt instance unless we have first resolved all instances that appear within it.

4. Repeat step 3 recursively, until every prompt instance appears at least once. If we are allowing different instances of the same prompt to contain different bindings, we have to continue to expand until either every remaining prompt instance has no prompts inside itself, or we detect an infinite descent (in which case we have to break).

5. Strike out duplicates. Each merged instance only appears once – the first place it occurs.

3.7 USING THE RESOLUTION OBJECT

The resolution object is a collection of prompts. It contains neither prompt objects, nor prompt instances, but instead something called a *prompt question.* Each prompt question represents a single question that might be given to the user. A prompt question is obtained by merging together prompt instances from the report.

We expect the user of a resolution to provide answers to all of the prompts in the resolution. We say that a prompt question is *closed* if the user has told us which answer to use for it. We provide several ways to access a prompt question from the collection: by index or by location.

3.7.1 Closed and Used prompts

One of the objectives of the resolution stage of report execution is to provide an answer to each prompt that occurs in the report instance. The resolution object collects together all of the prompts that appear in the report instance. There are several ways in which an answer can be bound to a prompt: -

1) The answer was known in advance. This happens if a report instance is executed using an old resolution object. We may find that a suitable answer already exists in the collection. Our normal behavior is to offer the previous answer to the user for further consideration, but if the "auto close effect" described in Section 3.2.1 is turned on, then we will accept the previous answer without offering it to the user.

2) The designer explicitly stated that a prompt should be closed, even if there was no previous answer. Again, setting the `Reuse` property does this. The designer can override the `Default` value on a prompt instance to make the instance default to a particular value. A designer might do this if she wants to reuse an existing application object, but also wants to fix the value of one of the prompts in the object. We covered this functionality in Section 3.3.4 above.

3) The client of the resolution object supplies an explicit answer to a prompt. We expect that this corresponds to a user selection in some prompt dialog, but of course all we know is that a function is called to set the answer. For all we can determine the client could have read the answer from a spreadsheet or from any other repository of data.

4) The client of the resolution object has three specialized ways to answer a prompt. The client can close the prompt to the `Previous` value of the prompt (if any is known), the client can close the prompt to its `Default` value, or the client can *cancel* the prompt. Canceling the prompt means that the client explicitly states that the prompt should not have an answer. It corresponds to the 'Empty' value of a variant data type.

5) Last of all the client of the resolution object may explicitly refuse to answer the question. We say that our client has *declined to answer*. The resolution object's response depends on the `Reuse` setting made when the prompt object or prompt instance was defined. The default setting of `Reuse` states that the resolution object should close a prompt to its `Default` value if the client declines to answer it.

The Boolean property `Closed` is used to record whether or not a property has been answered. It is set to 'True' once the prompt has been closed. In cases (1) and (2) above the prompt is placed in the resolution object with `Closed` already set to 'True' so we know not to re-prompt the user.

We also have the Boolean property called `Used`. This property is set to 'True' for a prompt question that corresponds to a prompt instance that appears somewhere in the report instance. It is possible for a prompt question to appear in the resolution object with `Used` set to 'False', because the resolution server will not remove prompt questions that it doesn't need. This is because we want to retain the previous answers for these prompt questions in case it later turns out that we do need them after all.

It is possible that a prompt instance appears in a report, but is still marked as not being `Used`. This happens if the import / export properties mean that another prompt instance is imported to replace the prompt instance. Again this only happens if a prompt question based on this prompt instance already happened to be in the resolution object. The resolution server will never add a prompt question to the resolution object unless it wants to use it.

3.7.2 Answering a prompt

In case (3) above a user usually answers the prompt by setting the `Answer` property. This will automatically set the `Closed` property to True. It will not change the `Used` property.

```
' Set the answer to the prompt to an explicit value (usually obtained from a user)
Prompt.Answer = 1997
```

The type of values that can be used depends on the type of the prompt. Some types of prompt cannot be set in this way. For these prompts the `Answer` property always returns an object of the correct type. The user sets the `Answer` by modifying the object it returns. See the chapter on types of prompts for details of this.

3.7.3 Special answers for a prompt

As we stated in case (4) above there are three special answers that are usually valid for a prompt question. The first is to *cancel the prompt*. This means that the user explicitly states that the prompt should have no answer. The user does this by setting `Answer` to 'Empty".

```
' If we want to cancel the prompt altogether
' This command will return an error if the prompt does not support canceling
Prompt.Answer = Empty
```

The value 'Empty' is acceptable to any type of prompt. However the `Reuse` property can be used to specify that a certain prompt instance has to be answered rather than canceled. In this case we will return an error when the user attempts to cancel the prompt. We consider canceling the prompt to mean that the user doesn't want to use the answer of the prompt at all. For example if the user cancels a Double prompt then the resolution server will remove all references to the prompt (by deleting branches of expressions etc.) rather than just substituting zero for the answer of the prompt. In some cases it is not possible to remove a reference to a prompt, in which case the resolution server will return an error when it tries to substitute an answer to the prompt. As far as possible the resolution server will detect this situation in advance, and set `Reuse` accordingly, but there are situations in which it is unable to do so. The designer can always set `Reuse` to forbid canceling a prompt.

The other special answers are to close a question prompt to its `Default` answer or to its `Previous` answer. The user sets the prompt to its `Default` or `Previous` answer by setting the `Answer` property to the value of the relevant property. These properties return 'Empty' if no value is recorded. This means that if the user attempts to set a prompt to its `Previous` (or `Default`) value, and there were no `Previous` (or `Default`) value, then the user will cancel the prompt.

```
' If we want to use the same answer as last time the prompt instance was encountered
' Prompt.Previous will be Empty if no previous value is known
Prompt.Answer = Prompt.Previous

' If we want to use the default answer from the prompt object
Prompt.Answer = Prompt.Default
```

We only store the answer for a prompt, not the way in which the answer was obtained. Suppose a user closes a prompt by setting it to its default value, and then the designer changes the default value for the prompt. If the user runs the report again, and closes the prompt to its previous value, the prompt will be closed to the old previous value, not to the current default value.

A prompt question has a Boolean property called `HasPrevious`. This property is 'True' if and only if the `Previous` property is really set to a previous value. This property allows a user to distinguish between `Previous` set to 'Empty' because there is no previous answer, and `Previous` set to 'Empty' because the user canceled the prompt last time it was encountered. There is not a 'HasDefault' property. We don't need it since there is no point in distinguishing between "no default answer" and "default answer is to cancel the prompt", because if we don't find a default answer we have to cancel the prompt anyway.

3.7.4 Declining to answer a prompt

A user can also decline to answer a prompt. This is not the same as canceling the prompt. Declining a prompt says that the user doesn't want to answer the prompt or is not available to answer the prompt. If a report is executed as a scheduled job, then the resolution server will automatically decline every prompt in the report, since there is no user to answer the prompt. Usually the resolution object will respond to a declined prompt by taking the `Default` value, or canceling the prompt if there is no `Default` value.

The designer can modify this behavior by changing the `Reuse` property for the prompt instance. The options available are to cancels the prompt, takes the `Default` value, takes the `Previous` value or return an error. See Section 3.2.1 for a complete list of settings.

A user can determine in advance which answer will be used if he declines to answer a prompt. It is the same value that the `Answer` property will return before `Closed` is set to 'True'. A user indicates that he wants to decline a particular prompt question, by setting the question's `Closed` property to 'True'.

```
' Mark the prompt as closed.
' The Reuse enumeration is used to calculate the answer
Prompt.Closed = True
```

3.7.5 Accessing by order

A client-side GUI wants to present the prompts in order. So we allow the client to access the resolution collection by index number. This access method shows all of the prompts in the resolution, whether or not an answer already exists for them, and whether or not they are actually used.

```
' Go through all of the prompts in the order we want to present them to the client
Dim Prompt As IDSSPrompt
For Each Prompt in Resolution
   Call AnswerPrompt ( Prompt )
Next Prompt
```

If the client uses a wizard like dialog to present these prompts to the user, we can suppose that a user might not answer them in order. So in general we allow the user to answer them in any order. There is an exception to this in the special case in which one prompt's answer is used to validate another prompt. In this case the prompts must be answered in the correct order since we cannot offer the correct choices to a user if they were answered in the wrong order. We use the `Incomplete` property and the `Locked` property explained in Section 3.7.7 to handle this situation.

We also provide a method called `NextPrompt` that returns the first unanswered prompt. This method will return the first useful prompt in the collection that has not been answered already. It would return NULL if all of the prompts in the collection have either been answered, or are not used. To be precise this method returns the lowest numbered prompt question which has `Closed` set to 'False', `Used` set to 'True' and `Incomplete` set to 'False'.

```
' Loop through all the prompts in the resolution that need to be answered
' Skip those that have already been answered / don't need to be answered
Set Prompt = Resolution.NextPrompt
Do Until Prompt Is Nothing
    Call AnswerPrompt ( Prompt )
    Set Prompt = Resolution.NextPrompt
Loop
```

The `NextPrompt` method will not return blank prompts inside draft prompts. This does not matter because when a user encounters a draft prompt in the resolution collection the user needs to answer all of the blank prompts in the draft prompt in order to answer the draft prompt. So if the user closes all of the top-level question prompts in the resolution, the user must have answered all of the blank prompts in the resolution as well. See Section 4.3 for more information on draft prompts and blank prompts.

The `NextPrompt` method takes an optional argument. This is a prompt question from the resolution object. If this argument is set then we will search for prompts after the given prompt, instead of starting from the beginning of the collection.

3.7.6 Location of a prompt question

The user may want to know where a prompt question came from. This means that we tell the client, which prompt instance(s) caused us to add the prompt question to the resolution. Each prompt instance is called a *location* of the prompt question. Since, as the name implies, a prompt question may correspond to several individual prompt instances, a prompt question may have several locations.

We expose a collection `Locations` of all of the locations that the prompt appears. Each individual location is presented as an application object (`Container`) and an index number (`Index`). The prompt question came from the prompt instance with the given index number in this application object. The `Prompt` property of the IDSSPromptLocation interface is a shortcut to the prompt instance.

```
' Return all the locations where Prompt is used
Dim Loc As IDSSPromptLocation
For Each Loc In Prompt.Locations
    MsgBox "Prompt is in " & Loc.Container.Name & " (position " & Loc.Index ") " & _
           "Prompt's Meaning is " & Loc.Prompt.Meaning
Next Loc
```

As we saw earlier (in Section 3.3.2) a designer can elect to include several instances of the same prompt in the same application object. So knowing the prompt object and the application object alone are not sufficient to identify the location. The `Index` property is essential to clarify which instance is intended.

Observe also that there may be several prompt questions that include the same location in their collection of locations. The example we gave in Section 3.5 shows how this can happen. In this

example we have two prompt questions of the Year prompt that both report "Instance (1) in SimpleInterest2" as one of their locations.

However it is not possible for there to be two prompt questions which both return exactly the same set of locations. If such prompts existed, our merging algorithm would have merged them into a single prompt question.

In fact due to the way we merge prompts the first location in the locations collection will always be unique in the sense that no other prompt question in the resolution object will list the same first location. See Section 3.7.8 for more information on the meaning of the first location of a prompt.

3.7.7 Incomplete prompts

Sometimes a prompt (the 'outer' prompt) cannot be answered because the prompt is dependent on another prompt (the 'inner' prompt). To be more precise this happens when the outer prompt uses the inner prompt to define its validation properties. We cannot determine whether an answer to the outer prompt is valid until after the user has assigned an answer to the inner prompt. When this happens we say that the outer prompt is *incomplete*.

We indicate this situation by means of a read-only Boolean property called `Incomplete`. Usually this property returns 'False'. However for an incomplete prompt question it returns 'True'. An incomplete prompt cannot be closed until after all its inner prompt(s) have been closed.

The resolution server must be used to change a prompt's status from incomplete to complete. This is because the way we remove incomplete prompts is to substitute the answers given to the inner prompts into the validation properties of the outer prompt. Only the resolution server has the power to perform substitutions of answers into prompts.

The concept of incomplete prompts is why the resolution object may override the validation properties of a prompt question. By using import / export it is possible to design a report instance in which the same prompt is resolved multiple times with different values substituted in for its inner prompts.

A resolution object usually presents inner prompts before outer prompts in its collection. (The only exception is if the inner prompts appear in a draft prompt). If a user examines the prompt questions in order, and closes and each one and then runs the resolution server again before going on to the next prompt, then the user will never encounter an incomplete prompt. However it is much more efficient for a user to close all complete prompts before passing the report instance back to the resolution server. Also we permit users to close the prompts in any order. So the `Incomplete` property is needed to cover the cases when we are unable to close a prompt because another prompt has not been closed yet.

```
' Scan through a resolution, looking for incomplete prompts
Dim Prompt As IDSSPrompt
For Each Prompt in Resolution
    If Prompt.Incomplete Then
        MsgBox "Prompt " & Prompt.Info.Name & " cannot be closed"

        ' The following line would return an ERROR "attempt to close an incomplete prompt"
        Prompt.Closed = True ' ERROR
    End If
Next Prompt
```

The converse to the Incomplete property is the Locked property. Again this is a read-only Boolean property. We set it to indicate that a prompt question cannot be modified because its Answer was substituted into another prompt. If we allowed a user to modify the Answer then the substitution would be rendered invalid. Again for reasons of efficiency we don't want to immediately fix up the other prompt whenever the user modifies the answer.

If a user wants to modify a Locked prompt, she should set the Closed property of the prompt to 'False'. The next time the user invokes the resolution server it will clear the Locked flag of any open prompt that it finds. This will always cause some other prompt to lose its validation, and be set to Incomplete.

When one prompt question is dependent on the other prompt question in this manner it will always be the case that one or the other of them cannot be modified. Either the outer prompt is Incomplete or the inner prompt is Locked.

3.7.8 Accessing by location

Although, as we described in Section 3.7.5, each prompt in the resolution exposes a collection of locations, it is in fact the case that we view each prompt as coming from a particular location, called its *primary location*. Primary locations are unlikely to be of interest to users; we use them to merge prompts together. Should a user care to discover the primary location of a prompt, it is always indicated by first location in the prompt's Locations collection.

There are two types of primary location: -

- *Global primary location.* This corresponds to prompt instances whose Merge property was set to true. A prompt is in the global location, if either it has no entries in its Locations collection, or the first instance in the collection has Merge set to 'True'. It is not possible for two instances of the same prompt object to appear in a resolution object, and both to be in the global location (because they would have been merged).

- *Local primary location.* This corresponds to a prompt instance from a specific application object. The instance is always the first location in the prompt's Locations collection. It always has Merge set to 'False'. The only way that a local prompt can list multiple locations is if the import / export feature was used to merge it with other prompt instances. Again a resolution object will never contain two prompts that give the same primary location.

A *secondary location* is any location of a prompt question other than its primary location. A resolution object records the secondary locations of all of its prompts, but they are not relevant to merging prompts together.

A user can find a particular prompt question in a resolution by using either a prompt object or a prompt instance as an index. For a prompt object the resolution object will return the corresponding prompt question whose global primary location corresponds to the prompt object. It returns the same thing for any prompt instance whose Merge property is 'True'. For any other prompt instance it returns the prompt question whose primary location is the given instance.

This functionality is needed by any component that wants to behave as if the answer in the resolution object was used instead of the prompt itself. It needs to be able to find which instance in the resolution corresponds to the external instance.

```
' Find the prompt assigned to the Year prompt instance in SimpleInterest
Set Prompt = Resolution ( SimpleInterest.Info.Prompts ( Year ) )
```

This type of look-up may fail ("object not found") in situations in which a cursory glance might lead one to expect it to succeed. Consider again the example in Section 3.5. In this example we have a prompt Year that appears as the first prompt instance in the metric SimpleInterest2. However using `SimpleInterest2.Info.Prompts(1)` as an index will fail.

The reason is that the report doesn't really contain this prompt instance. Both times it would have been used it was replaced by an imported prompt instance; the instances of Year on the template were used instead. The resolution server is clever enough to spot that the prompt whose primary location is the first instance in SimpleInterest2 is unused, and so it doesn't include it in the resolution object. Thus the look up fails. (Observe also that we can't just use the prompt question that replaced it – there are two of them and no reason to favor one of them over the other. The situation is ambiguous and so it is better to fail than choose in an arbitrary fashion.)

Looking up a prompt question by location will only return a prompt that appears in the top-level prompt question collection of the resolution object. A blank prompt, which only appears under its draft prompt, will not be found by this method.

3.7.9 Applying a resolution

The resolution object has a method `Apply` whose effect is to make a new object based on an original object and information contained in the resolution. The method behaves like the `Copy` method in that it makes a new local object based on the original object. The difference between this method and copy is that whenever a prompt is encountered in the original object we examine the resolution. If an answer is bound to a corresponding prompt instance in the resolution, we place the answer, not the prompt, in the new object.

```
' Create a new metric based on the original metric
Dim M As IDSSMetric
Set M = Resolution.Apply ( SimpleInterest )
```

If the original object didn't contain any prompts, which are answered in the resolution, then we return the original object. This functionality is essential for efficiency reasons – otherwise we would duplicate every application object in a report instance each time the user executed the report instance.

If we make a new object, the user can save it back to the metadata in the usual manner. We use this facility to implement "prompts as object definition templates" – it allows us to create an object based on an earlier object, with all of the prompts from the earlier object replaced from the resolution.

The copy that we make is actually made recursively. That is we also create copies of any dependent objects of the argument, if the dependents contain prompts. We have to do this in order to make an object that acts like the argument, in the context of all of the prompt answers in the resolution. In particular we may have to copy an object that doesn't contain prompts directly, because the object contains objects that contain prompts.

We embed the copies of the dependents in the new object that we make, so that a client, who is not interested in the dependents, does not need to worry about them. Observer that this gives a client a way to apply the resolution to objects in a different way to our behavior when the resolution is applied directly. For example if the resolution is applied to the template T2 in our example in Section 3.5, we will create a copy of T2, and two copies of SimpleInterest2. The two metric copies will differ, because they will have different values replacing their YearStop prompt instance.

If the user applies the resolution directly to SimpleInterest2 then we would not replace the first instance of Year at all, because the resolution object doesn't contain a prompt question that replaces that instance. The knowledge that it was imported from the instances of the prompt in the template is contained in the template, not the resolution object, so it is only used when the resolution is applied to the template.

The initial implementation of the `Apply` method requires that all of the prompts in the object appear in the resolution object, and have been closed there. Later implementations will relax this requirement.

3.8 EDITING A RESOLUTION OBJECT

We provide various methods on the resolution object that can be used to modify which prompts appear in the resolution object. A client can call these methods if a client wants to add or remove prompts directly from a resolution object.

We intend for clients to use these methods to edit resolution objects that they intend to save to metadata. It is not a good idea for a client to edit a resolution object during report execution, since the resolution server will not be able to detect the changes, and may become confused.

3.8.1 Collection methods

The resolution object is a collection. So we provide the usual methods to edit the collection. In particular we have `Clear` and `Remove` to take prompts out of the collection, and `Add` to put prompts into the collection.

`Clear` works in the usual way – it removes all of the prompts from a resolution. Remove is used to remove a specific member from the collection. The member can be referred to by index number, by passing in a reference to the prompt question that is to be removed, or by using any `Item` parameter that can be unambiguously matched to a particular prompt question.

The `Add` method can be used to add a new prompt question. The argument to Add can either be a prompt object (in which case we add a prompt question for `Merge` equals 'True' instances of the prompt) or a prompt instance. If the collection already contains a suitable instance then we will respond by returning the existing instance, not adding a new instance.

```
' Add a prompt question to the resolution based on the Year prompt object
' This instance is the one used for all Year prompt instances with Merge=True
Dim YearResolution As IDSSPrompt
Set YearResolution = Resolution.Add Year

' Can also add a prompt question based on a particular prompt instance
' This one is the one used for the prompt instance if it isn't merged with anything else
Dim YearStopResolution As IDSSPrompt
Set YearStopResolution = Resolution.Add YearStop
```

3.8.2 Combining with another resolution

We provide a special functionality for one resolution to combine its prompt questions with the prompts in another resolution. This functionality is provided by the `Combine` method. It creates new prompt questions in this resolution, based on those in the other resolution.

```
' Combine those prompts that appear in S, but do not appear in R into R
R.Combine S

' In fact the Combine method is just a shortcut for the following code fragment
Dim P As IDSSPrompt
For Each P In S
    R.Add P
Next P
```

3.8.3 The CleanUp method

A resolution object has a method `CleanUp` that takes no parameters. This method causes the resolution object to clean itself up by deleting all information stored in the object involved with executing a specific report instance. We only retain the answers given to the prompt questions. Specifically this method has the following effects: -

- All location information (except the primary location) is deleted.
- No prompt is marked as `Used`.
- No prompt is marked as `Closed`. The old answer for each prompt is moved to its `Previous` property.
- Any prompt whose validation properties refer to another prompt is marked as `Incomplete`. Any value previously substituted into the validation properties is deleted.
- Other internal storage used during resolution is freed.

A user should call this method after report execution is complete, if the user wants to save the resolution object used for report execution into the metadata. It is not necessary to clean up a resolution object before saving it to the metadata, but failing to do so will result in a substantial storage overhead, as it saves irrelevant information. Since there are legitimate reasons for a user to want to save a resolution object complete with execution information (e.g. the user wants to schedule the report to execute at a later date) the object server will permit a user to save a resolution object without cleaning it up first.

3.8.4 Resolving a specific object

We provide a functionality to use to resolve a specific object. This functionality is also used in report execution, when we apply it to the report definition object that acts as the starting place of the definition of the report. The operation of resolving an object means to walk through the object, and all of its known dependents, and add prompts to a resolution object for all of the prompts that we find. This process is performed by a special component called the *resolution server*.

Call the `Resolve` method to ask the resolution object to resolve an object. The resolution object will invoke the resolution server, to scan the argument, and its dependents, looking for prompts. We will add all the prompts discovered to the resolution object. The resolution object will be cleaned up before we add new information. The `Resolve` method takes an optional parameter, `CacheOnly`

that defaults to False. Setting this parameter to 'True' guarantees that no new objects will be loaded during resolution.

This functionality is essential if a client wants to use an object with prompts as a 'template' for building other objects. We expect the functionality to be used as described in the following code snippet. Suppose `MetricTemplate` is a metric object, which is believed to contain prompts.

```
' Subroutine to return the default metric based on a metric template
Public Sub DefaultMetric ( MetricTemplate As IDSSMetric ) As IDSSMetric

    ' We make a new resolution object to act as a scratch pad
    Dim Resolution As IDSSResolution
    Set Resolution = OS.NewObject DssTypeResolution

    ' We obtain a resolved list of prompts from the metric template object
    Resolution.Resolve MetricTemplate

    ' Keep on looping, in case the answers introduces more prompts
    Do Until Resolution.NextPrompt Is Nothing

        ' All we need to do, is to close all the open prompts in the resolution
        Do Until Resolution.NextPrompt Is Nothing
            Reolution.NextPrompt.Closed = True
        Loop

        ' Now we need to re-resolve the object (The 'True' here is for repeat resolution)
        Resolution.Resolve MetricTemplate, True
    Loop

    ' Now we are ready to make the new metric based on the MetricTemplate
    Set DefaultMetric = Resolution.Apply MetricTemplate

End Sub
```

The resolution action will fail if the resolution server detects some inconsistency in the way that the dependent objects are organized. The following situations will all generate an error; there may be others: -

- There is a cycle in the dependency graph. For example if an object A is defined in terms of an object B, and object B is defined in terms of object A then the resolution server will refuse to resolve any report which uses either object. Where possible the editors will reject objects that contain dependency cycles, but since we do not use a single transaction to save all objects involved in a report, it is possible to create this situation by editing A and B simultaneously.

- There is a cycle in the import / export relationships. It is an error if two mutually dependent objects both export and import the same prompt to each other. We require that import / export is used to allow a 'parent' object to take control of a prompt from its 'child'. If the import / export relationships contains a cycle we cannot determine which object is the parent and which is the child.

- There is a cycle in the 'incomplete' dependence between prompts. It is not acceptable for prompts P and Q to both use the other prompt as one of its validation properties. If such a configuration were permitted then both prompts would be marked as `Incomplete` and a user would be unable to close either prompt.

- There is a merging inconsistency involving prompts as validation properties for other prompts. Suppose that the answer of one (inner) prompt is used inside another (outer) prompt, either as a validation property, or as a blank prompt in a draft prompt. By using multiple instances and import / export it is possible for a user to contrive a situation in which we specify that the inner

prompt is asked twice, and that the outer prompt only asked once. This clearly cannot be resolved since we have a contradiction; the outer prompt must be asked once, but it must also be asked using two different values of the inner prompt.

- We also forbid the situation described in the previous paragraph for non-draft prompts when only a single instance appears of the outer prompt, even though the definition does not explicitly say that the outer prompt is asked only once. This situation is forbidden because it would be highly confusing for a user, who would be asked the same prompt twice with the same title and meaning. The fact that the two prompt instances do in fact differ (by the validation requirements we apply to the answer) would not be apparent to the user.

The last two situations are unlikely to occur unless a designer specifically sets out to contrive a situation that involves them. Any designer cunning enough to generate one of these errors is likely to understand what went wrong.

3.8.5 Resolving an object again

Our default behavior when a user asks to resolve an object is to clean up then resolution object before starting the resolution. However there are many situations in which an object cannot be completely resolved with a single execution of the resolution server. For example: -

- One of the prompts in the resolution object is incomplete. The resolution server needs to be invoked a second time to make it substitute in the answer to the inner prompts.
- The answer to one of the prompts refers to an object that did not previously appear as a dependent. We have to scan the new object, and its dependents for further prompts.
- A user closes an object valued prompt. The resolution server must be used to determine if any other prompts are exported from the newly closed prompt.

We provide an optional parameter `Repeat` that defaults to 'False'. A user should set this parameter to 'True' if the user wants to keep existing information in the resolution object. In the last Sub-section we showed an example of how this parameter is used.

```
' Now we need to re-resolve the object (The 'True' here is for repeat resolution)
Resolution.Resolve MetricTemplate, True
```

4. PROMPT TYPES

In this Section we go into detail about the various types of prompt. For each type of prompt we describe how to read and write the values assigned to the prompt. We also explain what validation properties are available for each type of prompt.

The type of a prompt is indicated by the `PromptType` enumeration. For every type of prompt an answer for the prompt is represented using a single VARIANT property. When the prompts answer cannot be expressed as a simple value, the property returns a suitable collection object that can be used to contain the answer. Since a prompt question has up to three answers associated with it, there are three properties available to hold an answer: -

- The default answer for the prompt is held in the property `Default`.
- The answer for a particular prompt instance is held in the `Answer` property.
- The previous answer for a prompt question is returned in the `Previous` property.

The following properties are available to validate the answer of a prompt. The meaning, if any, of one of these properties depends on the type of the prompt. For some types of prompts these properties are unused. These validation properties are all assigned at the prompt object level. They cannot be overridden by a prompt instance. We give these properties bland names, and make them Variants so that we can reuse them as different things in different contexts. We want to be able to add new prompt types without changing this interface.

- **Minimum.** (VARIANT) This property is always the same type as `Maximum` (or Empty), and is never bigger than `Maximum`.
- **Maximum.** (VARIANT) This property is always the same type as `Minimum` (or Empty), and is never smaller than `Minimum`.
- **Restriction.** (VARIANT) An entity that puts a restriction on the available answers. Typically a filter, a search object or an expression type
- **Origin.** (VARIANT) An object that acts as a starting point for finding objects that satisfy the prompt. Typically an attribute or a filter.

4.1 SIMPLE PROMPTS

A *simple prompt* is a prompt whose answer corresponds to a simple datatype. We define simple prompts for the most common Variant types. There is no need to define a simple prompt for those Variant types that are not used in the COM API.

4.1.1 Boolean

A *Boolean prompt* is a prompt whose answer has the type Boolean. It has no validation properties, since for the Boolean type eliminating a possible value is tantamount to specifying which value is returned. (There is no point is defining a prompt whose answer is True or False, but for which False is not a valid answer – since this means that the user is not being given a choice at all.)

Thus all that is necessary to define a Boolean prompt is to declare that the prompt is Boolean. As usual we define the title and meaning of the prompt, since these are the properties that distinguish between different Boolean prompts.

```
Prompt.PromptType = DssPromptBoolean
Prompt.Title = "Year - Absolute or Filtered"
Prompt.Meaning = "Do you want to see the result summed over all years (True) or just for the
    years in the filter (False)?"
```

4.1.2 Long

A *Long prompt* is a prompt whose answer has type Long. A prompt's designer can use the `Minimum` and `Maximum` properties to specify `Minimum` and `Maximum` values for a valid answer. These properties should either by set to Empty (signifying no minimum/maximum) or to a Long value. We want `Minimum` to be smaller than or equal to the `Answer` property, which in turn must be smaller than or equal to `Maximum`.

The following code shows a prompt to return an integer valued percentage. This means that integers in the range [0, 100] are accepted.

```
Prompt.PromptType = DssPromptLong
Prompt.Minimum = 0
Prompt.Maximum = 100
Prompt.Title = "Percentage"
```

4.1.3 String

A *String prompt* is a prompt whose answer is a string value. Our default behavior is to accept any string as a valid response. Assign integer values to `Minimum` or `Maximum` to specify a valid range for the length of the string.

For example the following code declares a string prompt whose sole validation constraint is that the empty string is forbidden. There is no upper limit on the length of the string.

```
Prompt.PromptType = DssPromptString
Prompt.Minimum = 1
Prompt.Maximum = Empty
Prompt.Title = "Enter Name"
```

4.1.4 Double

A *Double prompt* is a prompt whose answer has type Double. A prompt's designer can use the `Minimum` and `Maximum` properties to specify `Minimum` and `Maximum` values for a valid answer. These properties should either by set to Empty (signifying no minimum/maximum) or to a numeric value. We want `Minimum` to be smaller than or equal to `Answer`, which in turn must be smaller than or equal to `Maximum`.

The following code shows a prompt to return percentages. This means that doubles in the range [0, 100] are accepted.

```
Prompt.PromptType = DssPromptDouble
Prompt.Minimum = 0
Prompt.Maximum = 100
Prompt.Title = "Percentage"
```

4.1.5 Date

A *Date prompt* is a prompt used to specify a date. Hopefully this will be done using the VB Date type, but the final decision will depend on how the kernel team chooses to represent date values. We may need to introduce more validation properties (for example if we want a date prompt that accepts only midweek dates).

4.2 COMPLEX PROMPTS

A *complex prompt* is a prompt that cannot be expressed as a single value, but nevertheless is a fundamental COM object in the COM API. Most uses of prompts in the COM API are complex prompts.

4.2.1 DSS Object(s)

A *(DSS) Object prompt* is a prompt whose result takes the form of a set of DSS Objects drawn from the metadata. We use the `Minimum` and `Maximum` properties with integer values to specify an acceptable range for the number of objects in the answer. In particular a designer would set both of these properties to 1 to define a prompt for a single DSS Object.

Unfortunately specifying how many objects are acceptable does not provide sufficient information to validate an object prompt. We also need to know which objects are acceptable. Rather than define a new set of properties for the various ways of limiting an object choice we reuse the existing concept of a search object.

A *search object* is an object that is used to specify a search. It does this by containing a list of restrictions on objects (by type, by name, by description, by parent folder, by owner, by content etc.) An object prompt uses a search object to validate its answer. Only objects that would have been returned by the search are acceptable. The COM API will not automatically execute the search when a prompt containing it is resolved. A GUI might choose to execute the search, and then invite the user to choose from the results, but this strategy would not be advisable if (for example) the search merely specifies that the answer is a filter. A search object is passed to the prompt by setting its `Restriction` property.

A search object is a DSS Object in its own right, so it is saved separately into the metadata. A user can always embed the search object in the prompt object (or in the same container as the prompt object if the prompt object is itself embedded) if the user wishes to conceal the use of a separate object.

The following code defines an object prompt that only accepts a single filter taken from a specified folder (or one of its descendents). We embed the search object in the prompt (so it behaves as a single object).

```
' Create a search object to specify which objects are acceptable
Dim S As IDSSSearch
Set S = Prompt.Info.Embedded.Add DssTypeSearch
S.Types.Add DssTypeFilter
Set S.SearchRoot = TheFolder

' Now assign the properties on the prompt
Prompt.PromptType = DssPromptObjects
Prompt.Minimum = 1
Prompt.Maximum = 1
Set Prompt.Restriction = S
```

In order to use an object prompt we need to explain how to read and write an answer for the prompt. The VARIANT returned by all three value properties of an object prompt (`Default`, `Answer`, `Previous`) will either be empty (if no value has been specified) or it will contain an object reference (i.e. an IDispatch pointer) to a folder object.

The folder object (which will act like a search folder) will contain whichever objects the user or designer selected. In particular this allows a client to distinguish between the absence of the value (Variant will be empty), and the assigned value "no objects selected" (Variant will be an empty folder).

```
' This function prints out a message about the default value of its object prompt
Public Sub ObjectsPromptDefault ( P As IDSSPrompt )

    Dim D As Variant
    Dim F As IDSSFolder

    ' Check we were given a prompt, and that it is an objects prompt
    If P Is Nothing Then Exit Sub
    If P.Type <> DssPromptObjects Then Exit Sub

    D = P.Default
    If VarType ( D ) = vbEmpty Then
        Print "Prompt does not have a default value"
    Else
        ' There is a value, it has to be a folder
        Set F = D
        If F.Count = 0 Then
            Print "Prompt's default value is to select no objects"
        Else
            If F.Count = 1 Then
                Print "Prompt's default value is to select " & F(1).Name
            Else
                Print "Prompt's default value consists of multiple object"
            End If
        End If
    End If
End Sub
```

A user also needs to be able to assign a value to the prompt. There are two basic ways they can do this. One way is to assign something to the property as a whole. This is useful if the user has access to a single object that describes the selection. However if the user wants to select multiple objects, then the user has to use the `AddCopy` method of the folder returned by the property to add additional objects to the property. Of course the user can only amend the value if they obtained the prompt interface in a context which allows write access to it.

```
' Unset the default value (so the prompt no longer has a default)
Prompt.Default = Empty

' Set the default value to return no objects
Prompt.Default = Nothing

' Set the default value to return a specific object
Prompt.Default = MyFilter

' Note that setting the value to a folder will cause the prompt to return the folder
' as a single object.  It will not make the prompt return the objects in the folder
' The following line will cause the prompt to return a single object - the root folder
Prompt.Default = OS.Root

' In order to return multiple objects, first set the prompt to something other than Empty
Prompt.Default = Nothing

' Then read the folder from the prompt
Dim F As IDSSFolder
Set F = Prompt.Default

' Finally add the objects to the prompt
' Note since the Folder interface's Add method is used to create new objects you must use
      AddCopy instead.  The objects are not really copied.
F.AddCopy MyFilter
F.AddCopy MyMetric
F.AddCopt MyTemplate
```

4.2.2 Elements

An *elements prompt* is a prompt whose answer takes the form of a set of elements. An elements prompt is very similar to an objects prompt in the sense that the answer takes the form of a collection. We have similar issues – how to validate which elements may be put in the collection, how to read the contents of the collection, how to modify the collection.

As with the objects prompt, we use the `Minimum` and `Maximum` properties to specify a range of acceptable sizes for an element collection. As usual a user sets these properties to empty if the user doesn't care to set a restriction.

If the designer does nothing else then the prompt will accept any set of elements. Generally speaking we expect that the designer will want to limit the set of available elements in some way. It is not clear if there is a reasonable way to present an unrestricted choice to a user.

The simplest way to do this is to assign a value to the `Origin` property. This value is interpreted as describing the place from which we draw the elements. We can either use an Attribute (meaning that the elements are drawn from the given attribute) or a Dimension (meaning that the elements are drawn from an attribute in the dimension). A dimension object contains all the browse path information needed to allow the user to chose elements from the dimension.

In addition to containing relationships between their attributes, dimensions also contain filtering information. So a designer could create a dimension that specifies that only a subset of its elements are presented to a user. The dimension object also has provision to store multiple filters, and even filter templates (user types in first three letters of element etc.) to allow sophisticated control of element browsing.

This functionality would permit a designer to specify a prompt that consists of selecting elements from an attribute using a particular filter – the designer could do this by creating a dimension that contains just the chosen attribute. However to prevent the designer from having to create singleton dimensions in this way we permit the designer to set the `Restriction` property to a filter object

when the Origin property is an attribute. This means that the prompt only contains elements from the attribute that satisfy the filter. Since a filter can be defined using an explicit list of elements, this allows a designer to specify exactly which elements are available for the prompt. However using a dimension is the only way that a designer can modify the default browse forms collection in the attribute.

We also accept the configuration of a dimension in the Origin property and an attribute in the Restriction property. This means that only elements from the given attribute are acceptable, but the user should be asked to use the given dimension to select the elements. The user can browse through other attributes in the dimension to reach the requested attribute, but cannot select elements from these attributes.

The following code sample shows a prompt for a non-empty list of elements. The attribute is given. We also specify a filter to use to restrict the choice. Any filter (that the engine can relate to the attribute) can be used here.

```
' Prompt on all the elements in an attribute
Prompt.PromptType = DssPromptElements
Set Prompt.Origin = MyAttribute

' Require the user to give an entry
Prompt.Minimum = 1

' Add a filter restriction (could be list of elements / qualification on attribute form /
      even a qualification on a related attribute)
Set Prompt.Restriction = MyFilter
```

Accessing one of the value properties (Default, Answer or Previous) for the prompt object always returns either empty (if the value property has not been assigned) or an element collection (containing whichever elements were selected). We do this even if the collection contains only zero or a single element. This enables a user to distinguish between an unassigned value and a value that has been assigned to the empty collection without requiring a user to handle too many cases when examining one of these values.

The following code sample shows how a user can investigate a value property (in this case the Answer) of a prompt. We print out a message indicating whether or not the prompt has an answer, and if it does have an answer, how many elements it contains. Note that the Answer property is populated in a resolution object before the prompt is closed in order to indicate what the answer would be if the user declines the prompt.

```
Public Sub ElementsPromptAnswer ( P As IDSSPrompt )

    Dim A As Variant
    Dim E As IDSSElements

    ' Check we were given a prompt, and that it is an elements prompt
    If P Is Nothing Then Exit Sub
    If P.Type <> DssPromptElements Then Exit Sub
```

```
    A = P.Answer
    If VarType ( A ) = vbEmpty Then
        Print "Prompt does not have an answer"
    Else
        ' There is a value, it has to be a element collection
        Set E = A
        If E.Count = 0 Then
            Print "Prompt's answer is to select no elements"
        Else
            If E.Count = 1 Then
                Print "Prompt's answer selects a single element"
            Else
                Print "Prompt's answer selects multiple elements"
            End If
        End If
    End If
End Sub
```

Last of all a user needs to be able to assign a new value to an elements prompt. As with the objects prompt the user can do this either by assigning a complete value to the property, or by retrieving an element collection from the property and then using the normal collection methods to modify the collection. The complete assignment can be to empty (to unset the property), to Nothing (to make the property into an empty collection, to a single element (to make the collection contain just this element) or to an existing collection of elements (the entire collection is copied into the prompt instance).

```
' Unset the Answer property
Prompt.Answer = Empty

' Set the Answer property to indicate that no elements were chosen
Prompt.Answer = Nothing

' Set the Answer property to the first element in MyAttribute
Prompt.Answer = MyAttribute.Elements ( 1 )

' Set the Answer to be all of the elements in MyAttribute
Prompt.Answer = MyAttribute.Elements

' Set the Answer to be every other element in MyAttribute
' This code is using late binding; its more efficient to cast Prompt.Answer to an
        IDSSElements interface, so VB knows which method to call in advance
Prompt.Answer.Clear
Dim I As Long
For I = 1 To MyAttribute.Elements.Count Step 2
    Prompt.Answer.Add MyAttribute.Elements ( I )
Next I
```

4.2.3 Expression

An *expression prompt* is a prompt whose answer takes the form of a COM API expression. We use expressions to define both filters and metrics. A prompt for an expression is also useful to define concepts like a metric qualification.

Using an expression prompt represents a situation in which we allow the user to build any expression that they like. The only restrictions we impose on the user are a simple depth requirement, and a restriction on the type of expression that we allow. We expect that a standard expression editor will be used to prompt a user to answer an expression prompt.

For an expression prompt we use the `Minimum` and `Maximum` properties to record a range for the depth of a valid expression. These allow a designer to specify "how big" the expression is allowed

to be. The depth of an expression is the number of IDSSOperator nodes encountered on the longest path from the root of the expression to a leaf. Thus an expression has depth zero if it consists of a single leaf node, depth 1 if it contains a single operator (as in an Abell metric qualification "metric < 32"), depth 2 if it contains an operator nested within an operator (e.g. "metric < metric2 + 1") and so on. We say that an empty expression has depth –1.

An expression prompt uses the `Restriction` property to record which type of expression it represents. Assign this property with a value taken from `ExpressionType` enumeration. This enumeration, which remains to be defined, contains entries like DssExpressionFilter or DssExpressionMetricQualification and so on. Each parameter corresponds to a different type of expression that the parser is able to recognize and validate.

```
' Define a prompt that takes a non-empty metric expression
Prompt.PromptType = DssPromptExpression
Prompt.Minimum = 0
Prompt.Restriction = DssExpressionMetric
```

A value property (`Default`, `Answer` or `Previous`) either returns empty if the value is unset, or returns the IDSSExpression interface (if the prompt is assigned). As usual this allows the user to distinguish between the absence of a value, and the value which represents an empty expression.

```
Public Sub ExpressionPromptPrevious ( P As IDSSPrompt )

    Dim A As Variant
    Dim E As IDSSExpression

    ' Check we were given a prompt, and that it is an expression prompt
    If P Is Nothing Then Exit Sub
    If P.Type <> DssPromptExpression Then Exit Sub

    A = P.Previous
    If VarType ( A ) = vbEmpty Then
        Print "Prompt does not have a previous answer"
    Else
        ' There is a value, it has to be an expression
        Set E = A
        If E.Root = Nothing Then
            Print "Prompt's previous answer is an empty expression"
        Else
            If E.Root.Type <> DssTypeOperator Then
                Print "Prompt's previous answer is an expression with a single leaf"
            Else
                Print "Prompt's previous answer has depth at least one"
            End If
        End If
    End If
End Sub
```

Users assign an expression value to a property in the usual way. They would assign Empty to it to unset the property, 'Nothing' to set it to the empty expression, and an existing expression to replace the entire expression. Generally we expect a user to edit the expression directly, by reading an IDSSExpression interface from the value property, and then editing the expression object in the same manner as is used to construct metrics or any other expression based object.

```
' Unset the answer
Prompt.Answer = Empty

' Set the answer to be an empty expression
Prompt.Answer = Nothing

' Alternatively we can edit the expression directly
' In this case we make it an expression that contains a single string
Dim E As IDSSExpression
Set E = Prompt.Answer
E.Add ( DssTypeConstant ).Special.Value = "My Expression"
```

4.3 DRAFT PROMPTS

A *draft prompt* is a type of prompt which presents itself to a client as a collection of other prompts. We refer to the prompts in a draft prompt as *blank prompts* or simply *blanks*. A blank prompt is usually one of the simple or complex prompts that we outlined in the proceeding sections, although it can also itself be a draft prompt. A blank prompt does not carry enough context information to uniquely identify it to the user. When a user is answering blank prompts the user needs to know both which blank prompt he is answering, and which instance of the draft prompt question it comes from.

There are two benefits of using draft prompts. The major disadvantage of draft prompts is that they are more complex to use, and to understand than other types of prompts: -

- Draft prompts allow a designer to connect together several prompt objects into a single larger prompt. The user is prompted for all of these connected prompt objects as a single prompt. The user can answer the blank prompts as individual prompts, or answer the draft prompt as a single operation, in which case all the blank prompts are also answered.

- Draft prompts permit a designer to present prompts to the user with more context information than just the collection of prompt locations.

The collection of blank prompts in a draft prompt is presented in two ways. One way is as an ordered collection of prompt questions. The other way is in the form of a (read-only) DSS Object, or fragment of a DSS Object, called the draft prompt's *origin* that contains the blank prompts. The answer of a draft prompt consists of the origin with the answers to all of the blank prompts filled in.

If there are two or more non-merged instances of a draft prompt, then there will be two identical instances of the blank prompts that it contains. The resolution object will distinguish between these instances solely by which draft prompt instance they are below – there is no where for the designer of the report to place a separate `Title` and `Meaning` property for these blanks, since they both were generated from the same prompt instance. This is not a problem, since a client application will present them to the user as part of the draft prompt and not as separate prompt instances.

For the moment we only have one type of draft prompt, namely an expression draft prompt. We will add other types of draft prompt if we find a clear need for them.

4.3.1 Using draft prompts

Draft prompts are just a class of prompt types. So the COM API determines whether or not a prompt is a draft prompt by examining the prompt's type. For example the type `DssPromptExpressionDraft` is a type of draft prompt.

When defining a draft prompt use the `Origin` property to define the prompt's draft. The draft prompt appears to be an instance of whatever type of object, or object fragment the draft prompt represents. The designer should populate this object in the usual way, using normal interfaces, and prompt instances.

When populating the `Origin` object, the design should use other prompts to indicate blanks in the draft. There is no point in designing a draft prompt whose `Origin` object does not contain any other prompts, since this would mean that there is nothing left for the user to fill in. The blank prompts will appear in the draft prompt's instance collection.

When the resolution object populates its `Answer` property, it will copy over the fixed structure from the `Origin` property. For example if the `Origin` property contains an expression node, then the same node will be duplicated in the `Answer` property. When the `Origin` contains a prompt instance, the `Answer` will instead contain a prompt question. The `Prompts` collection of the draft prompt's question acts as a shortcut to all of these prompt questions. These questions are the blank prompts that define the draft prompt.

Both the `Merge` property and the import / export mechanism can be used to specify that a blank prompt is the same as an ordinary prompt that is available directly from the resolution collection. If this happens the resolution object will only make one prompt question, but the question can be accessed either directly from the resolution object, or from the prompt question that represents the draft prompt.

However if a blank prompt has `Merge` set to 'False', and no other prompt is imported to replace it, then it will not appear directly in the resolution collection. This is the behavior that sets the draft prompt apart from other types of prompt. The resolution object will create special prompt question for the blank prompt, which is only accessible from the draft prompt's question.

The user can close the draft prompt by closing each blank prompt in the draft prompt. The user either walks through the structures in the `Answer` property, or goes to the `Prompts` collection to obtain a list of blank prompts. Once the user has a blank prompt the user can close it using any one of the normal techniques. The answer to each blank prompt must satisfy the validation restrictions of that blank prompt.

4.3.2 Closing a draft prompt

There is nothing in a draft prompt other than its collection of blank prompts. A draft prompt is closed if and only if all of the blank prompts in the draft prompt are closed. Normally a user answers a draft prompt, by separately closing all of the blank prompts in the draft prompt. As soon as the user closes the last blank prompt, the `Closed` property of the draft prompt will set to 'True'.

However a user can also close a draft prompt using a single operation. When the user does this, the resolution object will automatically close all of the blank prompts. The following table describes how it does this.

User action on draft prompt	Effect of action on blank prompts	Situations in which we return an error
Cancels	All blank prompts are canceled, even if they were already closed, and even if their `Reuse` property says that they cannot be canceled.	Action fails only if the draft prompt cannot be canceled.
Sets to `Default`	Each open blank prompt is closed to its `Default` value,o r canceled if it has none.	Action fails if there is a blank prompt that has no default value,a nd which cannot be canceled.
Sets to `Previous`	Each open blank prompt is closed to its `Previous` value. If it has no `Previous` value then it is declined.	The action fails if a blank prompt's `Reuse` property says that the blank promptc annot be declined.
Declines to answer	Determines an action (previous / default / cancel /b reak) for the draft prompt. Then applies the action as described above.	Action fails if either the draft prompt's Reuse property says it cannot be declined, or if one of the blank prompts cannot be answered in the prescribed manner.

It is possible for a blank prompt to be incomplete, because it requires some other prompt to be closed in order to define the validation restrictions of the blank prompt. It is not possible for a draft prompt to be incomplete, since it doesn't have any validation restrictions of its own.

Observe that a draft prompt does not have a default value, or a previous value of its own. Instead it uses the values for each of its blank prompts. This means that both the `Default` and `Previous` properties of a draft prompt are set to 'Empty' on prompt objects and prompt instances. On a prompt question they should return structures showing what would happen if the user were to select the `Default` or `Previous` value. In particular we must be able to distinguish between the cases "`P.Answer = P.Default`", "`P.Answer = P.Previous`" and "`P.Answer = Empty`".

4.3.3 Draft prompts example

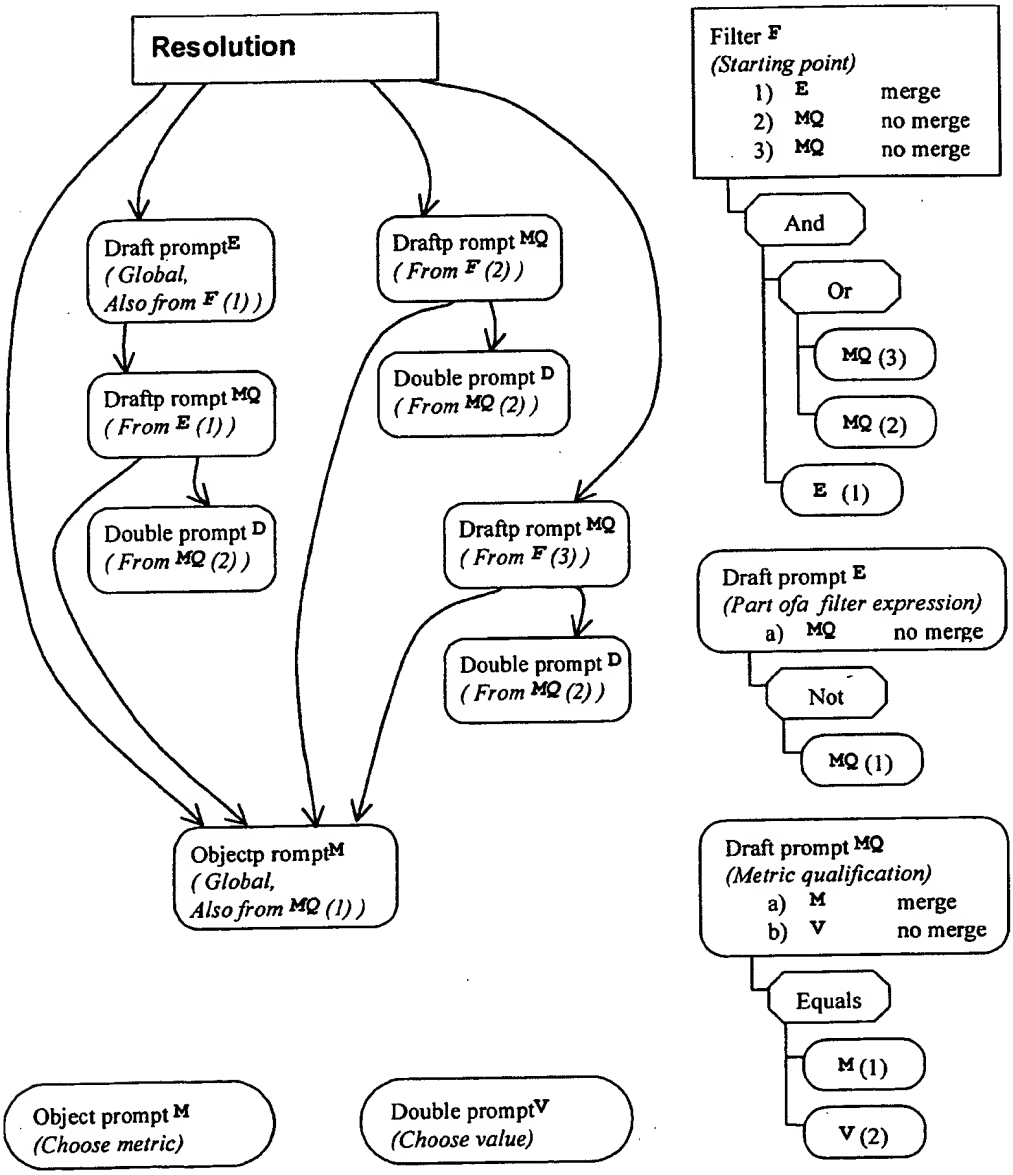
The following diagram shows an example of what happens when draft prompts are resolved. It shows the definitions of five objects – a filter called F, and four prompt objects, which consist of two draft prompts, called E and MQ, a metric prompt called M and a double prompt called V. Each object shows its definition, and also the prompt instances, if any, it contains.

We then show the resolution object that would be generated if the filter were resolved. Each oval below the resolution object represents one of the eight prompts that will be given to the user in order to answer all the prompts in the filter.

A normal prompt (like M and D) doesn't contain any other prompts, but the draft prompts contain other prompts. It is possible for one draft prompt to be nested inside another draft prompt. In the diagram the instance of the E prompt contains an instance of MQ, which in turn contains an instance of M and an instance of D. To simplify the diagram we have not drawn the expression trees in the resolution object, but in fact each draft prompt exposes its children via an expression tree as well as via a collection.

Normally a prompt question's primary location is sufficient to identify to which prompt question we are referring. For example MQ is asked three times, once for each usage of MQ in the definitions. However when a prompt is below a draft prompt the primary location may not suffice. In this

example the D prompt is asked three times, each time with the same primary location. The only way to distinguish between each occurrence is to see which draft prompt contains it. Since the M prompt's `Merge` property is 'True', this prompt is only asked once. The prompt question can be accessed directly from the main collection in the resolution object, and also from each instance of MQ.



4.3.4 Expression draft

An *expression draft prompt* is a draft prompt whose answer is an expression. It can be used to prompt for an entire expression, or to prompt for a sub-expression of a larger expression. It differs from the usual expression prompt because the designer of an expression draft prompt must provide

an `Origin` expression. This expression contains prompts to indicate blanks that we want the user to fill in when the draft prompt is executed.

We expect the expression draft prompt to be used to define something like a simple metric qualification prompt. The designer of the prompt can specify the format of the expression. The GUI can then recognize whether or not it is appropriate to use a special metric qualification dialog to ask the user to answer the prompt. (We expect the GUI to attach a special DSSProperty to a prompt to indicate whether or not it meets the requirements of such a dialog.)

`Origin` is the only validation property used for an expression draft prompt. As explained in Section 4.3.1 this property is used to define the draft of a valid answer. A designer inserts prompts in the `Origin` to indicate blanks in the draft. During report execution the user must supply an answer for each blank prompt. Unless `Merge` is set to 'True' for a blank prompt, or the import / export mechanism is used, the blank prompt must be answered once for each question that appears in the resolution object based on the draft prompt.

An example should make this clearer. Here we will define a metric qualification prompt. This prompt is based on three other prompts: "SelectBaseMetric", "ChooseOp" and "ChoosePositive".

SelectBaseMetric prompts the user to select a metric from a folder of metrics. We want to do this once during report execution, and use the same metric in the template, and in the metric qualification. ChooseOp selects a single operator object. ChoosePositive selects a floating-point number, that must be larger than zero. We want these prompts to be treated simply as 'blanks' in the definition of the metric qualification. If the metric qualification appears twice then they should be asked twice.

```
' First we set the type of the prompt
Prompt.PromptType = DssPromptExpressionDraft

' Now the Origin validation property contains an expression
Dim E As IDSSExpression
Set E = Prompt.Origin

' Create 3 nodes in the expression - answers to the prompt will contain these 3 nodes
Dim OpNode As IDSSOperator
Dim MetNode As IDSSShortcut
Dim ConstNode As IDSSConstant
Set OpNode = E.Add DssTypeOperator
Set MetNode = OpNode.Folder.Add DssTypeShortcut
Set ConstNode = OpNode.Folder.Add DssTypeConstant

' The metric node takes its value from the metric prompt
' As always the prompt will default to being asked once per report
' Since the prompt is shared we use its built in Title / Meaning
Set MetNode.Prompt = SelectBaseMetric

' The operator node will use the operator prompt
' We want this prompt to be treated as a blank in the draft
Set OpNode.Prompt = ChooseOp
OpNode.Prompt.Merge = False
OpNode.Prompt.Title = "Metric qualification operator"
OpNode.Prompt.Meaning = "The relationship operator =,<,> etc."

' The constant will use the prompt for a positive number
' Again we do not want this operator to be merged with its other instances
Set ConstNode.Prompt = ChoosePositive
ConstNode.Prompt.Merge = False
ConstNode.Prompt.Title = "Metric qualification value"
ConstNode.Prompt.Meaning = "The metric's value will be compared with this +ve number"
```

The three value properties (`Default`, `Answer` and `Previous`) of a question node based on an expression draft all return either 'Empty' or an IDSSExpression object. In the first case it indicates that the value is not set. In the second case the interface is the root of the expression. The expression can be examined in exactly the same way as it is examined for an expression prompt. The expression cannot be edited – its purpose is to show the user the relationship between the blank nodes in the expression.

Now suppose that a user wants to answer our expression draft prompt. The user needs to answer the single question based on the SelectBaseMetric prompt. It doesn't matter if the user takes the question directly from the resolution object or from the question based on the draft prompt. The resolution object will only make one question, and will return it from multiple places.

The user also needs to close the ChooseOp and ChoosePositive questions. These questions can only be found under questions based on the draft prompt. The user can either extract them from the `Answer` property of the draft prompt or use the `Prompts` collection on the draft prompt. If the draft prompt appears several times in the resolution object, then the blank questions need to be answered separately for each appearance of the draft prompt.

```
' Find the question based on the Prompt in the resolution object
Set Prompt = Resolution ( Prompt )

' Read the expression from this prompt
Set E = Prompt.Answer

' Take references on the three nodes in the answer
' We know what they will be, since it has to follow the draft
Set OpNode = E.Root
Set MetNode = OpNode.Folder(1)
Set ConstNode = OpNode.Folder(2)

' To find out which metric the user choose
' We assume the user answered it directly from the resolution object
Dim M As IDSSMetric
Set M = MetNode.Prompt.Answer

' To obtain the question prompt that specifies the function choice in the operator
Dim P As IDSSPrompt
Set P = OpNode.Prompt

' To set the function chosen by the user (assuming the symbol is bound to fn object)
Set P.Answer = FunctionGreaterThan

' To obtain the prompt that specifiec the validation restrictions on the constant
Set P = ConstNode.Prompt

' Again we set the value of the node in the usual way
P.Answer = 1.15
```

4.4 EXOTIC PROMPTS

An *exotic prompt* is a type of prompt that is created to cover a very specific situation. Generally speaking each exotic prompt is only used in one or two places. They are too specialized for widespread use.

We do not propose any exotic prompts at this time, since we cannot design an exotic prompt until the concept that we want to prompt for has been finalized. However the following objects are candidates for exotic prompts: -

- Metric dimensionalities (IDSSDimty). But do we want to make this into a separate DSS Object? Metric ingredients probably make this type of prompt unnecessary.
- Template sort. But are we going to split the template?
- Subtotals on a template. Again, are we going to split the template?

It is not necessary to use an exotic prompt in order to prompt for concepts covered by the exotic prompt. For example a 'choose attribute' prompt could be used inside an IDSSDimty to change the dimensionality; similarly a template subtotal could be defined to offer the user a list of metric subtotal objects.

However if we want the functionality that the system brings up the 'dimensionality editor' if there is such a thing, then we need to have a type of prompt that says, "edit a metric dimensionality as a whole". Exotic prompts are not needed to prompt the user to edit one or more of the properties that make up a metric dimensionality.

4.5 SUMMARY OF PROMPT TYPES

Here we list a summary of all of the available prompt types, and the acceptable data types for each type of prompt for the various validation properties. Note that in this list we have included several types of prompts that we did not define in the proceeding sections. These are suggestions are only included to allow for future expansion of the prompt concept.

Name	Answer	Minimum	Maximum	Origin	Restriction
DssPromptBoolean	True or False	Empty	Empty	Empty	Empty
DssPromptLong	Long	Long	Long	Empty	Empty
DssPromptString	String	Long	Long	Empty	Empty
DssPromptDouble	Double	Double	Double	Empty	Empty
DssPromptDate					
DssPromptObjects	ObjectInfo or Folder	Long	Long	Empty	Search
DssPromptElements	Elements or Element	Long	Long	Dimension or Attribute	Attribute or Filter
DssPromptExpression	Expression	Long	Long	Empty	Expression Type
DssPromptExpressionDraft	Expression	Empty	Empty	Expression	Empty
DssPromptFilterDraft					
DssPromptMetricDraft					
DssPromptTemplateDraft					
DssPromptDimty					
DssPromptSort					
DssPromptTemplateSubtotal					

5. USING PROMPTS IN PROMPTS

As we observed in Section 3.4.3, it is possible to place a prompt object within another prompt object. In that Section we were only concerned with how export and import worked when one prompt was used inside another prompt. In this Section we will discuss how one prompt can make use of another prompt.

These is nothing particularly special about using one prompt inside another prompt. The procedure is the same as when a prompt is used in any other application object. The object that wants to use the prompt exposes a property (called `Prompt` if it only takes one prompt like IDSSConstant, or called something like `XXXXPrompt` if it is the prompt for the property `XXXX`). The object that wants to use the prompt assigns the prompt to the prompt-valued property.

5.1 MEANING OF A PROMPT INSIDE A PROMPT

The prompt-valued properties of a prompt are defined at the prompt object level. It is not possible to see that one instance of a prompt does contain prompts, and another instance does not. The import / export mechanism can be used to modify how the prompts inside a prompt are resolved.

When a prompt appears inside another prompt, it forces the inner prompt to be answered before the outer prompt. The value of the inner prompt is then used to validate the outer prompt when it is answered. It is an error to mutually embed two prompts in each other. Prompts defined in this way will fail their validation test. If a user executes a report containing mutually recursive prompts, then report execution will fail at the resolution stage.

It is also possible for one prompt to use another one indirectly. This happens if a DSS object assigned to one of the prompt's properties contains prompts itself. For example a user could place a prompt in the filter assigned to the `Restriction` property of an elements prompt. Again we require that the prompts be evaluated in order; otherwise we cannot generate the correct list of elements for the user to use to select the outer prompt.

The mechanism of *incomplete prompts* described in Section 3.7.7 is used to enforce this dependency of one prompt on another. It allows the resolution object to refuse to close the outer prompt until the inner prompt has been closed.

5.2 SPECIFIC PROMPT VALUED PROPERTIES

There are four prompt-valued properties in a prompt object, which correspond to the four validation properties of a prompt. These allow a user to define a prompt whose value depends on another prompt that has to be answered first. We do not let a user supply a prompt for the answer value of a prompt – since we have already defined numerous mechanisms by which a user can specify that two prompts have the same answer.

We have the following four prompt-valued properties: -

- `MinimumPrompt`
- `MaximumPrompt`
- `RestrictionPrompt`
- `OriginPrompt`

The following example shows something a designer could do with these properties. We define a prompt so that its validation depends on another prompt.

First we define a prompt "SelectFilter" in which the user is asked to select a filter from a folder "StoreFiltersFolder" of filters. The designer of the prompts knows that all of the filters in the folder correspond to different restrictions on the Stores attribute "AttributeStores". The following code shows how to define this prompt.

```
' Make the prompt
Dim SelectFilter As IDSSPrompt
Set SelectFilter = OS.NewObject DssTypePrompt

With SelectFilter
    ' Its an objects prompt
    .PromptType = DssPromptObjects

    ' we want to select exactly one object
    .Minimum = 1
    .Maximum = 1

    ' embed a search object, specifying the correct folder
    .Restriction = SelectFilter.Info.Embedded.Add DssTypeSearch
    .Restriction.SearchRoot = StoreFiltersFolder
End With

' Just in case the folder contains something other than filters
Restriction.Types.Add DssTypeFilter
```

Now the designer wants to make a prompt that asks the user to select a single Store *from the Stores in the filter chosen by SelectFilter*. The only way to do this is to make the new prompt dependent on the old one. We also see now why the locking mechanism is essential; there is no way to determine if an element is valid until the user has chosen the filter. The following code shows how to define the new prompt.

```
' Make the prompt
Dim SelectElement As IDSSPrompt
Set SelectElement = OS.NewObject DssTypePrompt

With SelectElement
    ' Its an elements prompt
    .PromptType = DssPromptElements

    ' we want to select exactly one element
    .Minimum = 1
    .Maximum = 1

    ' the element must be a Store
    .Origin = AttributeStores

    ' use a prompt instead of setting the Restriction property to a filter
    .RestrictionPrompt = SelectFilter
End With
```

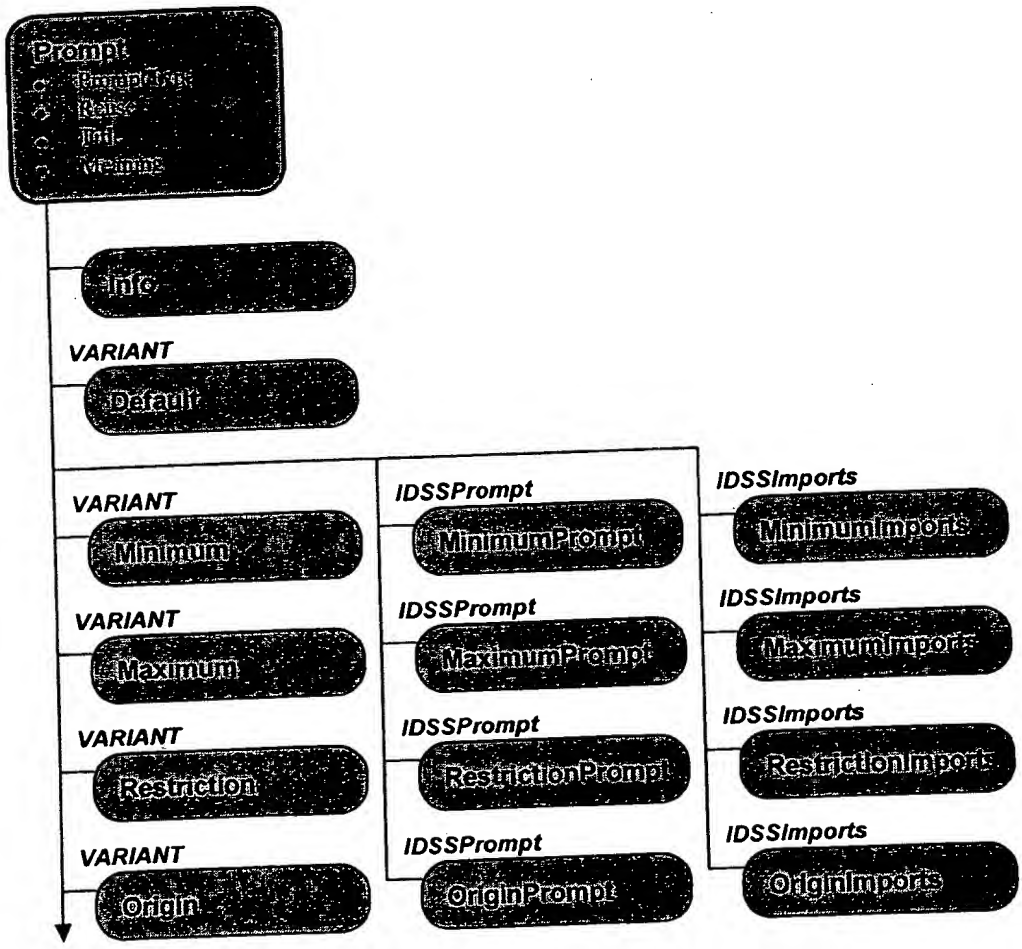
A designer can achieve much more complex relationships between prompts by permitting one prompt to indirectly refer to another prompt. The concepts however are the same.

6. OBJECT MAPS

This Section contains diagrams showing the relationships between the COM API objects used to manipulate prompts. We show three different object maps, for the three different ways in which we can use prompts. In each case we show only the properties that are relevant to the particular case.

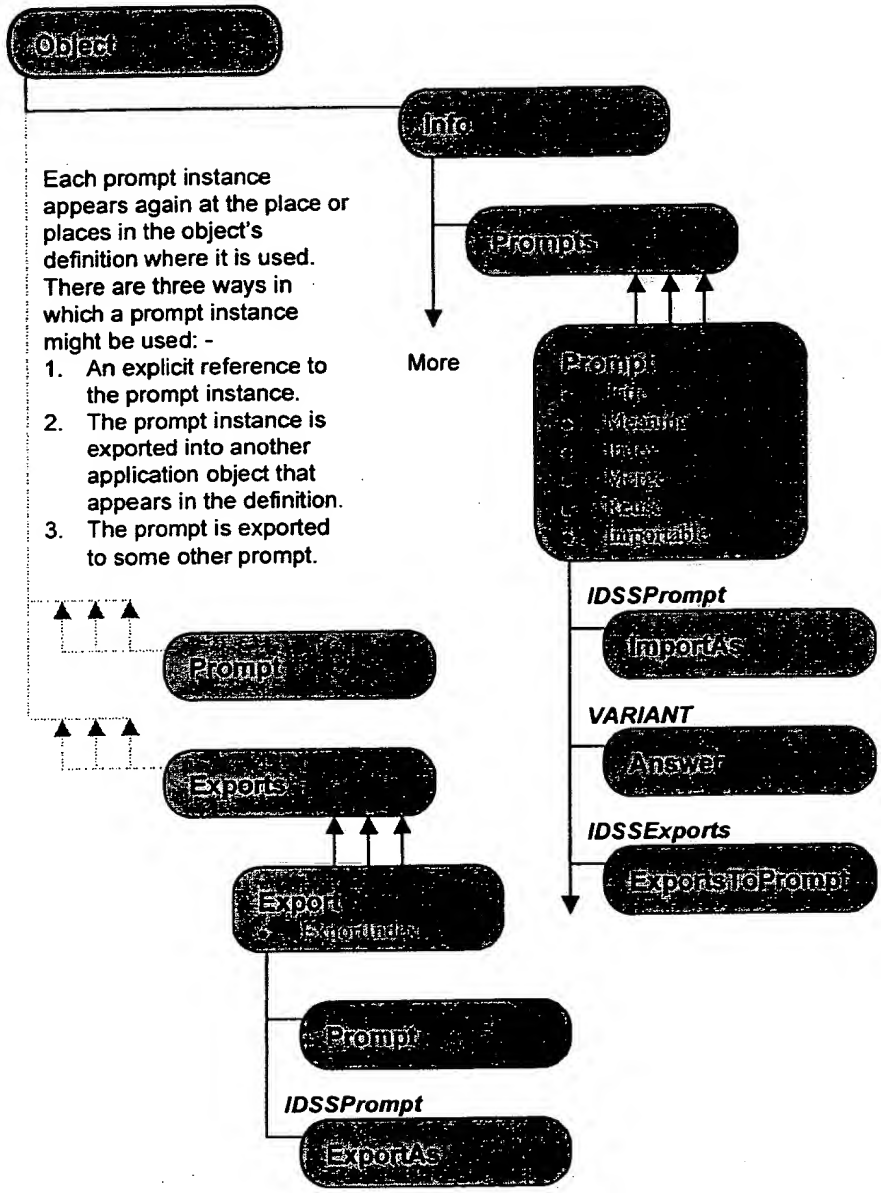
6.1 THE PROMPT OBJECT

In this map we show the properties on the prompt interface that are used when defining prompt objects. The prompt interface has other properties, but they are not used to define prompt objects.



6.2 PROMPT INSTANCE

This diagram shows the ways in which a prompt instance can be used on an object. We only show those properties on the prompt interface that can be assigned when the prompt is used as a prompt instance. We show where the prompt is used in an object.



6.3 RESOLUTION OBJECT

Here we show the properties of the IDSSPrompt interface that are used when it is accessed from the resolution interface. In this case we have properties to show the current and former value of the prompt. We also get a collection of locations where the prompt was used.